NetView for AIX

SC31-8164-00

Programmer's Guide

Version 4

SC31-8164-00

NetView for AIX

Programmer's Guide

Version 4

Note

Before using this product, read the general information under “Notices” on page ix.

First Edition (July 1995)

This document applies to IBM NetView for AIX (feature 5608), which is a feature of SystemView for AIX (5765-527). IBM NetView for AIX runs under the AIX Operating System for RISC System/6000 Version 3 Release 2 (5756-030) or Version 4 Release 1 (5765-393). This product is based, in part, on Hewlett-Packard Company's OpenView product.

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality.

A form for your comments is provided at the back of this document. If the form has been removed, you may address comments to:

IBM Corporation
Department CGMD
P.O. Box 12195
Research Triangle Park, North Carolina 27709
U.S.A.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Notices

Trademarks

AIX	APPN	IBM
InfoExplorer	MVS/DFP	NetView
NETCENTER	PS/2	RISC System/6000
RT	SystemView	System/390

About This Book

This book is for anybody who wants to write application programs to run with the IBM* NetView* for AIX* program.

Who Should Use This Book

Use this book if you will be writing programs that you will integrate with the NetView for AIX program, or if you will integrate other programs with NetView for AIX. You should have an understanding of the AIX operating system and be familiar with the C programming language.

Before writing applications to run with the NetView for AIX program, you should understand the program from the user's point of view. Familiarize yourself with the NetView for AIX program by using it and by reading the *NetView for AIX User's Guide for Beginners* and other manuals listed in the bibliography .

How to Use This Book

Use this book, in conjunction with the *NetView for AIX Programmer's Reference* and the *NetView for AIX Application Interface Style Guide*, to design, code, and integrate applications that will run with the NetView for AIX program. This guide is divided into three parts:

- Getting Started with NetView for AIX Programming
- Working with the NetView for AIX Graphical User Interface
- Using the NetView for AIX Management APIs

Each part contains several chapters, which address specific programming tasks. You should read all of the first part, and look at the other parts to determine which are relevant to your application. Each part begins with a detailed table of contents for that part.

Using the Man Pages

This guide will give you an understanding of the programming techniques you will use. Specific details, such as the syntax of routine calls, are provided in the *NetView for AIX Programmer's Reference*. Much of the information in the *NetView for AIX Programmer's Reference* is also available through the **man** command, which you can enter on your aixterm screen. The information seen in this way is called *man pages* or *reference pages*. After you use this guide to determine which routines or data structures you will use, you can find their exact specifications in the *NetView for AIX Programmer's Reference* or in the man pages. If you are viewing this book online using the DynaText browser, you can click on routine names that are displayed in blue to see the description of that routine in the online *NetView for AIX Programmer's Reference*.

Highlighting and Operation Naming Conventions

The following highlighting conventions are used in this book, with the noted exceptions:

Bold	Identifies commands and shell script paths (except in reference information), default values, user selections, daemon paths (on first occurrence), and flags (in parameter lists).
Italics	Identifies parameters whose actual names or values are to be supplied by the user, and terms that are defined in the following text.
`Monospace`	Identifies subjects of examples, messages in text, examples of portions of program code, examples of text you might see displayed, information you should actually type, and examples used as teaching aids.

The NetView for AIX operation naming convention used in this book shows the location of the operation in relation to the menu bar or context menu. The naming convention follows the format shown in this example:

```
Monitor..Network Configuration..Addresses
```

In this example, `Monitor` is a menu bar or context menu option, `Network Configuration` is an operation available from the Monitor submenu, and `Addresses` is an option that is available when you select Network Configuration.

Some operations require you to make selections from several layers of submenus before you reach the submenu containing the operation.

What Is New in This Book

Chapter 20, “Using the General Topology Manager” on page 315 and Chapter 21, “Communicating with the General Topology Manager” on page 343 have been updated to add more details about the NetView for AIX Open Topology MIB and to describe the new General Topology Manager (GTM) API routines.

Chapter 4, “Integrating Your Application with NetView for AIX Security Services” on page 43 has information about the new NetView for AIX security services.

Appendix, “Migrating Version 3 Applications” on page 359 has information about migrating Version 3 applications to Version 4

Where to Find More Information

The Internet Request for Comments (RFC) documents listed are shipped on the NetView for AIX program installation media and are installed in the /usr/OV/doc directory.

The following sources provide specific information that is not documented in the NetView for AIX Version 4 library:

- The Memo to Users provides additional information about the NetView for AIX program.
- The online help facility provides task, dialog box, and graphical interface information to help you use this program.

Part 1. Getting Started with NetView for AIX Programming

Chapter 1. Introducing the NetView for AIX Program

The NetView for AIX program is a powerful, flexible tool for managing networks. Because the world of network management is so dynamic, the NetView for AIX program provides several mechanisms with which you can extend its function or tailor it to your specific needs. This guide explains how to perform these tasks.

The Network Management Environment

Modern computer networks have grown and diversified to the point where they are a different sort of enterprise from computing environments of the past. No longer do networks consist of a collection of “dumb” terminals, all linked to one controlling computer, all made by the same vendor and speaking the same language. Some of the distinguishing characteristics of modern networks are:

- They are widely dispersed geographically.
- They include hardware and software from numerous vendors.
- They may use public telephone lines and other nonproprietary data links.
- They support an unprecedented volume of traffic.
- They include devices that can act autonomously, rather than being controlled by a master computer.
- They are subject to frequent changes in the status of the nodes that make up the network.
- They have a set of common functional requirements, but many networks require specialized capabilities.

The growth in the number and size of networks, and in their importance to many businesses and industries, has created a need for sophisticated management applications. These applications monitor and control networks to optimize their effectiveness as communications tools.

Because networks are no longer proprietary, standard protocols, which enable devices from different vendors to communicate with one another, have been defined by organizations such as ISO (International Standards Organization). These include TCP/IP (Transmission Control Protocol/Internet Protocol), CMIP (Common Management Information Protocol), and SNMP (Simple Network Management Protocol).

In addition to standard protocols, network management applications require standard ways to describe and interact with the resources that they manage. In order to shield management applications from having to know the implementation of every device in the network, an object-oriented approach is used.

The Object-Oriented Approach to Network Management

In the object-oriented model, each entity on the network is represented as a managed object. Each object is serviced by an agent, which is responsible for maintaining the data associated with the object, by responding to requests issued by management applications, or managers. These requests might involve reporting data values, setting or resetting status bits, or changing numeric or character values. Agents are also responsible for sending events, or traps, to managers to notify them of noteworthy conditions related to the agent's object.

The object-oriented approach has several characteristics that work well with the needs of network management applications:

- Management functions can be distributed among network nodes, rather than concentrated in one manager station.
- The management application does not need to know the definition of each object on the network.
- Each managed object, through its agent, is responsible for the maintenance of its own data and for responding to requests issued by managers.
- It supports asynchronous processing, in which a manager does not have to wait for a response from one request before processing or making another.
- It allows network objects to emit notifications, called events or traps, of network changes that may be of interest to managers.

Understanding NetView for AIX Terms

The following terms and concepts will be used throughout this Guide. A working knowledge of how to use the NetView for AIX program will also be beneficial when reading this manual.

Object An object is an internal representation of a logical or physical entity or resource that exists somewhere in a computer network. An object is made up of a set of fields that specify all the characteristics of the object. Programmers can create objects, delete objects, or modify the fields within an object. Programmers can retrieve the list of all fields associated with an object and search the object database for objects that contain specific field values. Examples of resources represented by objects might include:

- A computer node
- A software process on a computer
- An IP network

Note: The term *object* has a special meaning in the context of the XOM API. In that context it is called an *OM object*. Object without a modifier has the meaning described here.

Field Fields are the building blocks from which objects are constructed. Fields have one of the following data types: 32-bit integer, Boolean, character string, or enumerated value. An integer or character-string field can contain a single data element or a list of data elements of the same type; a Boolean field contains a single indicator, and an enumerated field contains one enumeration.

The NetView for AIX object database manages all object and field information for the NetView for AIX program. Developers can access the object database through the NetView for AIX programming libraries. See Chapter 7, “Creating and Using Objects and Fields” on page 79 for more information on objects and fields.

Map A map is a collection of NetView for AIX objects and their relationships. A map contains a subset of all the objects in the NetView for AIX object database. Different maps can display different management domains, or they can provide different presentations of the same domain.

Users, not applications, create maps. Users can create several maps, and they can control which applications operate on the various maps. While users create maps and define their scope, applications dynamically update maps to reflect the state of the management environment.

Submap A submap is a collection of related symbols that are displayed in a single graphical window. A submap provides a view into the map object space. Each submap displays a different perspective of the information in the map, with the submaps typically organized in a hierarchical fashion.

The most common method users employ to navigate through submaps is by double-clicking the mouse on symbols called explodable symbols. Double-clicking on an explodable symbol will cause a submap to be displayed if a submap is associated with the object represented by that symbol. The submap contains additional symbols which describe, in more detail, the object associated with the explodable symbol. The object associated with the explodable symbol is called the parent object. The submap that is displayed by double-clicking on the symbol associated with the parent object is called a child submap. For more information on working with submaps, see Chapter 9, “Creating and Using Submaps” on page 131.

Symbol A symbol is a graphical representation of an object as it appears on a submap of a particular map. Symbols are presentation elements; objects are underlying database elements that represent real-world network elements. Several symbols can represent the same object, even when the symbols are on different submaps.

A symbol can be either explodable or executable. When a user double-clicks on an explodable symbol, the child submap associated with that symbol will be displayed. When a user double-clicks on an executable symbol, the program associated with that symbol will be run.

Even though symbols represent objects, symbols can have some of their own characteristics beyond those of the object they represent. These characteristics, or attributes, can vary between the different symbols representing a particular object. For more information on these attributes, see Chapter 8, “Creating and Using Symbols” on page 99.

Selection List
The selection list is a list of objects that correspond to symbols selected by the user. The selection list is one of the primary ways that users pass arguments to NetView for AIX applications. See “Using the Object Selection List” on page 69.

Manager A manager is a software application that monitors and controls a network.

Agent An agent is a software application that is responsible for reporting on and maintaining the data pertaining to one or more objects. The interactions of managers and agents are described in “Transactions between Managers and Agents” on page 174.

Event The NetView for AIX program uses two distinct types of events. A map event is a notification issued because of a user or application action that affects the status of the current map or of the NetView for AIX graphical interface. A network event is a message sent by an agent to one or more managers to provide notification of an occurrence affecting a network object. In a network managed with SNMP, network events are called traps.

Filter An event filter is a specification used by an application to indicate which network events it wants to receive. Events may be filtered by node, frequency, event type, and various other criteria. Chapter 19, “Filtering Network Events” on page 307 describes event filtering in more detail.

Dialog Box
A dialog box is a window placed on the screen by a manager application to enable a user to specify details of a request. The NetView for AIX program may require your application's cooperation in processing user requests involving dialog boxes. More information on dialog boxes is provided in “Using Dialog Boxes” on page 74.

Registration Files
Registration files are configuration files that NetView for AIX reads when it is started. These files can be used to define many features of your application. The NetView for AIX program uses the following registration files:

- Application Registration Files
- Field Registration Files
- Local Registration Files
- Symbol Type Registration Files
- Security Registration Files

API An API (Application Programming Interface) is a library of routines that developers can use to construct applications. API routines are available to work with many areas of the NetView for AIX program. More information about APIs is provided in “NetView for AIX Application Programming Interfaces” on page 8. Specific details are provided in the *NetView for AIX Programmer's Reference* and in the online man pages.

The NetView for AIX End User Interface
The NetView for AIX program provides an advanced user interface that provides a graphical view of the network. The NetView for AIX program displays system and network management information through OSF/Motif ** windows. These windows contain submaps that show views of the network. Icons represent system or network elements, and lines represent the connections between elements. One of the primary ways for users to interact with the NetView for AIX program is to manipulate the objects represented in the window. The NetView for AIX EUI

API is a library of routines that manipulate the graphical user interface. These routines are the subject of Part 2, “Working with the NetView for AIX User Interface” on page 57.

Developing Applications for NetView for AIX

The NetView for AIX program uses an event-driven programming model. Programs developed under an event-driven model look fundamentally different from traditional programs. An event-driven program is a collection of routines that declare an interest in specific events generated by the user or system. These supporting routines are called as user and system events occur, and they perform the program's primary tasks. The program's main routine simply performs initialization, associates routines with particular events, and then enters a loop that waits for and processes events. The routine that is called when an event occurs is known as a callback routine. The declaration of a routine's interest in a particular event is referred to as callback registration. The portion of the main routine that waits for and processes events is called the event loop.

Developers construct NetView for AIX applications using two basic components: registration files and the NetView for AIX Application Programming Interface. The API is a library of routines that perform many programming tasks.

NetView for AIX Registration Files

Registration files are an easy way to define the static information about your application and the data with which it works. The following types of registration files are used:

Application Registration File
The Application Registration File (ARF) contains information about your application: its place in the NetView for AIX menu structure, the types of objects it can manipulate, the location of help information, and much more. Chapter 3, “Creating and Using the Application Registration File” on page 27 describes the ARF.

Field Registration File
The Field Registration File (FRF) includes definitions of new types of fields. See “Using the Field Registration File” on page 80 for more information.

Symbol Type Registration File
Use a Symbol Type Registration File (STRF) to create new symbols for your application to use. “Defining Symbols with Symbol Type Registration Files” on page 109 describes how to create new symbols.

Local Registration File
You should create a Local Registration File (LRF) for each agent that you create or install. You can also create LRFs for any daemons that you create. “The Local Registration File” on page 19 describes the LRF format.

Security Registration File
Version 4 of NetView for AIX provides you with a security service based on GSS-API authentication between clients and servers. Through the NetView for AIX security server, network administrators can control the access different users have to NetView for AIX features, including applications integrated into NetView for AIX. Users are assigned to security

'groups', and the administrator determines access privileges for different users and security groups. When you design your application, you need to decide whether you are going to exploit NetView for AIX security APIs to manage security.

To take advantage of the security feature, your application must use the Security Registration File (SRF) to register a security profile with the NetView for AIX security server.

NetView for AIX Application Programming Interfaces

The NetView for AIX program provides Application Programming Interfaces (APIs) that help you create applications for the NetView for AIX environment without learning the details of data structures, input/output requirements, and other low-level program components. This section contains a brief description of these APIs.

The XMP API

The NetView for AIX program manages objects in a TCP/IP environment that uses the SNMP and IP protocols for communication. It also supports the CMIP protocol and CMIS services over the TCP/IP protocol (CMOT), providing a way to manage OSI networks from an SNMP environment.

The XMP API masks the differences between SNMP objects and CMIS objects. It provides service primitives that correspond to the SNMP and CMIS services, so you can write an application to handle communication from either an SNMP agent or a CMOT agent. XMP employs the X/Open OSI-Abstract-Data Manipulation (XOM) API for general data manipulation.

For CMIS requests, the pmd daemon provides transparent control and initiation of all associations. Because SNMP is connectionless, the communications infrastructure hides the details of managing time-outs and retries for SNMP requests.

The XMP API is described in more detail in Chapter 13, “Using the XMP API” on page 213.

The SNMP API

You can develop SNMP-based network management applications with the NetView for AIX program's SNMP API. These applications can use the SNMP services and understand the SNMP MIB format. This API does not enable you to develop SNMP agents, other than supporting trap generation.

The NetView for AIX SNMP API also provides access to the NetView for AIX program's event filtering capabilities. By using filtering services provided through nvsnmp, applications can use event management services that would otherwise be available only through the XMP API. They can also control the number of traps an application receives, and reduce overhead in the application.

The NetView for AIX SNMP API is described in more detail in Chapter 17, “Using the NetView for AIX SNMP API” on page 281.

The OVuTL API

Because the interaction among managers and agents is often complex, it is important to be able to document and review the sequence of actions leading up to an unexpected behavior. The OVuTL API provides access to the NetView for AIX program's nettl tracing and logging facility. The nettl facility traces and logs the activity of network applications and formats the binary data to make it more useful for problem determination. For more information about the OVuTL API, see “Integrating Your Application with Logging and Tracing” on page 17.

The OVsPMD API

The OVsPMD API provides a way for an agent you develop to cooperate with the NetView for AIX process management daemon, ovspmd. For more information about the OVsPMD API, see “Process Management” on page 18.

The End User Interface API

The NetView for AIX end user interface (EUI) API enables you to develop user interfaces and integrate them with the NetView for AIX program.

The NetView for AIX EUI API provides routines to handle many aspects of programming, including:

- Connecting to the graphical interface
- Handling errors
- Creating objects and their component fields
- Creating and modifying submaps
- Creating and modifying symbols
- Verifying changes a user makes to a map
- Dynamically reconfiguring the graphical interface's menu bar, object menus, and Tool Window
- Processing callbacks.

The Event Filtering API

An NetView for AIX operator can define filters to limit both the number of events displayed in the event window and the number of traps that will be converted to alerts and sent to NetView. The Event Filtering API permits your application to access existing filters and to create new filters.

See Chapter 19, “Filtering Network Events” on page 307 for more information on the event filtering API.

The General Topology Manager API

The General Topology Manager (GTM) API enables you to communicate with the NetView for AIX program in order to store and present information about non-IP networks and devices. The GTM is described in Chapter 20, “Using the General Topology Manager” on page 315 and Chapter 21, “Communicating with the General Topology Manager” on page 343.

Figure 1 on page 10 illustrates how the major NetView for AIX APIs interact with your application.

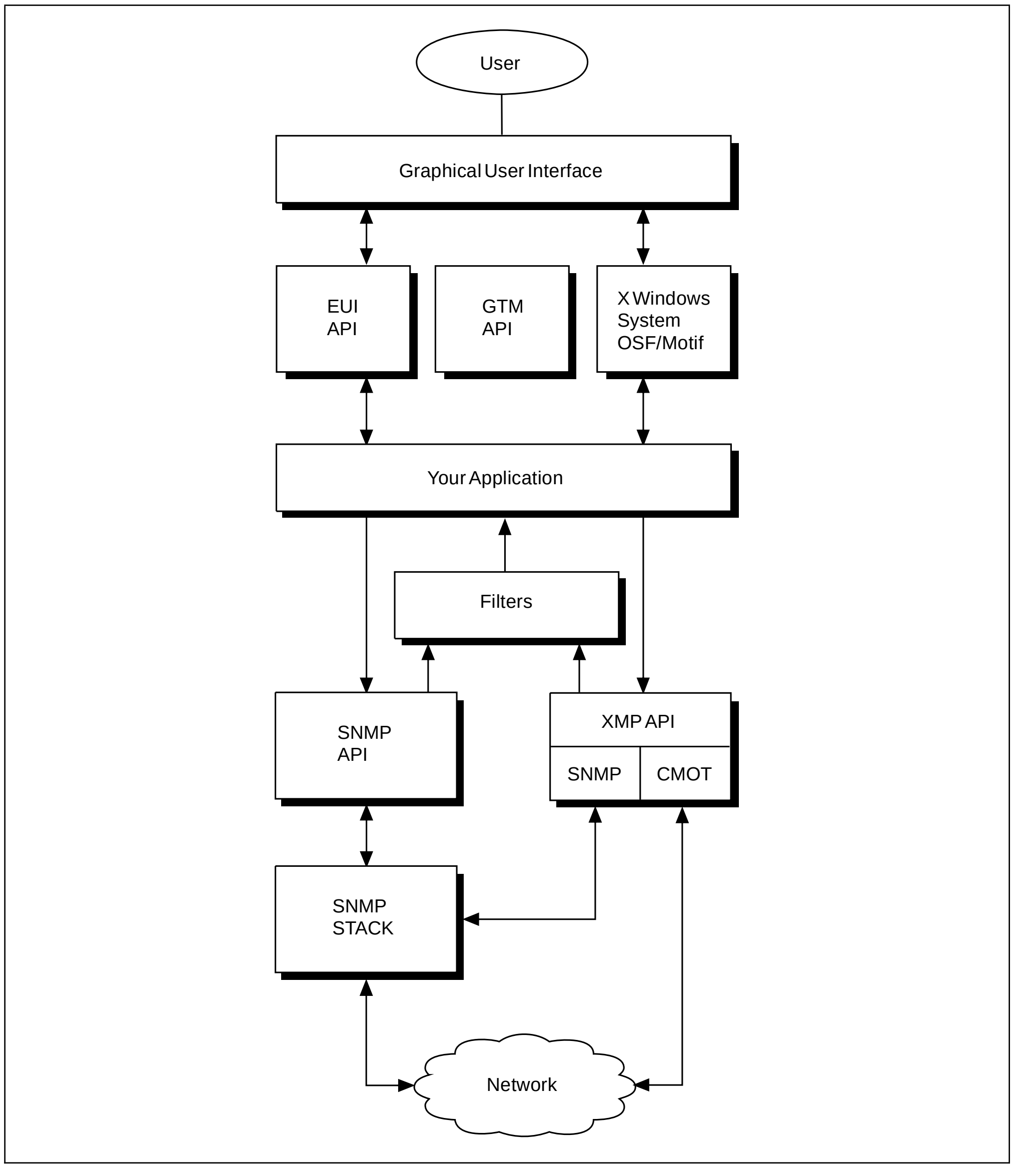

Figure 1. Using NetView for AIX APIs with Your Application

The NetView for AIX Help System

The NetView for AIX program provides a comprehensive help system to assist users with their tasks. Each application should provide its own help information, which the NetView for AIX program integrates into a common help system with a common user interface.

For Version 3, the NetView for AIX program implemented a new hypertext help system. Version 4 continues to use this help system. Links between help panels enable users to see information on related topics, and links from help panels to online books enable users to pursue more details than are included in help panels. Application developers can still use the ovhelp tool, as described in Chapter 5, “Designing Application Help” on page 49, to present help information.

Developers can integrate help information into the NetView for AIX program in a variety of ways. In most cases, you can construct files containing help information and place them in specific directories. The NetView for AIX program automatically integrates the help information in those files into the NetView for AIX help system. In other cases, you may need to call routines from the NetView for AIX EUI API to integrate your information into the NetView for AIX help system. See Chapter 5, “Designing Application Help” on page 49 for more information on integrating help information into the NetView for AIX program.

Designing Consistent Applications

As an application developer, you must make many choices when designing your application. Some of those decisions affect the physical appearance or behavior of the application. In order to provide consistency between different applications, consult the *NetView for AIX Application Interface Style Guide* as you develop your application. The style guide provides recommendations to help you ensure application consistency. For example, the style guide describes how applications should construct dialog boxes and how applications should use buttons and other user interface controls.

Developing Applications for a Client/Server Environment

A distributed client/server environment may or may not have a large impact on the design and packaging of your application, depending on what the application does. The client/server enabling of NetView for AIX provides the ability to distribute graphical applications (and the memory and CPU processing requirements associated with them) to client machines in the network. These clients connect to a server (the main management workstation) to receive topology information and other types of network management information.

Packaging Implications

If your application is launched from the menu bar and will be installed on every client, then the new client/server environment may have no impact on your application. Applications that require both a GUI and supporting daemons may require that the GUI be installed on clients and that the daemon be installed only on the server. Typically, GUI or end user interface functions should be viewed as client functions, and should be packaged to install on a client. Daemons should be packaged to install on the server. If you are unsure where a function belongs, package it to install on clients, since all workstations, including the server, have client functionality.

API Implications

The NetView for AIX API calls have been modified to hide the details of a client/server environment from the calling application. Applications that make calls to the OVw, OVsnmp, nvSnmp, and XMP APIs do not need to be changed to run in a client/server environment. API calls that involve communication with a daemon (either on a client or on the server) will automatically direct the call to the appropriate machine. It will be transparent to an application on a client whether an API call is crossing the network to a server or being made locally to a daemon that resides on the same machine. “Using APIs in a Client/Server Environment” on page 169 has more information about API support in a client/server environment.

Mount Implications

Much of the configuration information is shared between clients and the server through NFS mounts. Vendor MIBs are installed on the server machine. When a MIB is loaded, NetView for AIX makes the MIB available to clients by NFS mounting the /usr/OV/conf directory in read-only mode on the client machines. Any changes to the /usr/OV/conf directory on the server are visible to the clients.

Daemon Implications

Because daemons may be distributed, it is wise to not assume anything about the location of daemons in the network. For example, an application running on a client should not assume that there is a netmon daemon running on the client. Similarly, an application should not try to read the trapd.log file or other outputs from daemons unless the application is specifically designed to be a server function. Trap forwarding is another area that is sensitive to the location where an application is running; unless the application is specifically a server application, do not design the application to forward traps to LOOPBACK, because they will be lost on client machines. Applications that can run anywhere in the network should forward traps to the NetView for AIX server.

Network Communication Implications

Applications that use IPC calls for communication between parts of the application may need to be modified to use RPC or TCP ports as their communication mechanism.

Chapter 2. Integrating Your Application with NetView for AIX

This chapter describes the ways that applications can be integrated into the NetView for AIX program. By understanding the possible levels of application integration, you can determine which activities, and which chapters in this manual, apply to your application. In addition to the chapters suggested for each type of application, be sure to read the information presented later in this chapter on process management and tracing and logging.

Types of Applications

The NetView for AIX Application Interface Style Guide groups applications into three types:

- Drop-in applications
- Tool applications
- Map applications

These classifications are based on how tightly the application integrates with the NetView for AIX program. The characteristics of these application types are described in this chapter.

Regardless of what type of application you are developing, you should be familiar with the Version 4 NetView for AIX security feature. See Chapter 4, “Integrating Your Application with NetView for AIX Security Services” on page 43 for an introduction to NetView for AIX security and to understand the steps for integrating your application with NetView for AIX security.

Drop-In Applications

Drop-in applications are applications that integrate with the NetView for AIX program only through the menu bar. They are typically stand-alone applications that are integrated into the menu structure for user convenience. Drop-in applications do not use an Application Programming Interface (API).

To integrate drop-in applications into the NetView for AIX menu, you will create an application registration file, which controls where a menu item is placed in the NetView for AIX menu structure. The application registration file (ARF) also defines the command to be executed when the user selects the menu item. When the user selects the menu item, the application is invoked using the command specified in the ARF.

Drop-in applications can take advantage of the NetView for AIX environment. Use entries in the application registration file to configure the application so that it is invoked with NetView for AIX environment variables as arguments (for example, the selection name of the currently selected object). You can also configure the ARF File entry to allow the command to be executed only if the selected objects have particular characteristics. For example, you can specify that your application will be invoked only if the object is a node but not a gateway. Chapter 3, “Creating and Using the Application Registration File” on page 27 provides a complete description of the application registration file.

Examples of Drop-In Applications

One simple but useful example is to provide access to the telnet program through the NetView for AIX graphical interface. The user might select a node on a map, then select a telnet menu item from the NetView for AIX menu structure or tools window. The application registration file could specify that the telnet program be invoked with the IP hostname of the selected object as the argument.

Consider another example in which a standalone X-Windows-based program is added to the NetView for AIX menu structure. Assume that the X-based program retrieves the current disk system usage values from a remote system and displays them in a graph. The ARF might specify that the IP hostname be used to identify the remote system. The ARF might also specify that the command be invoked only for those systems with particular capabilities (for example, an object with the *node* capability field set).

Drop-in applications can integrate help information into the **Help** menu item on the NetView for AIX main menu bar. To integrate drop-in application help information, you place an appropriately formatted help file into a predefined directory, and identify that file in the application's ARF. The NetView for AIX help system automatically shows the help information to the user through one of the menu items in the NetView for AIX menu bar.

Chapters about Drop-In Applications

You can integrate drop-in applications into NetView for AIX using the information presented in:

- Chapter 3, "Creating and Using the Application Registration File" on page 27
- Chapter 4, "Integrating Your Application with NetView for AIX Security Services" on page 43
- Chapter 5, "Designing Application Help" on page 49

Tool Applications

Tool applications typically are written explicitly for integration with the NetView for AIX program and use the NetView for AIX Application Programming Interfaces (APIs). The *NetView for AIX Application Interface Style Guide* refers to applications that provide their own interface as application tools and application subtools. An application tool is a set of integrated functions within an application that is presented to the user in one or more windows. Application subtools are specific functions that can be carried out from an application tool. Applications that present their functions through tools and subtools are called *tool applications*.

Tool applications have the following characteristics:

- They are accessible through NetView for AIX menus or executable symbols. Once invoked by the user, tool applications usually provide their own user interface (for example, an application-specific menu structure).
- They can perform actions on objects that exist in the map.
- They use a limited set of NetView for AIX API routines.

Tool applications are integrated into the NetView for AIX program more tightly than are drop-in applications. Tool applications use the NetView for AIX registration files and usually provide some form of help access through the NetView for AIX help system. Tool applications also use the NetView for AIX API to some degree. Tool applications typically do not modify the contents of a map.

Note: Do not confuse tool applications as described here with applications that are placed into the tools window. If you create a tool application, you might want to add it to the tools window, but you are not required to do so. Similarly, you might want to add a new tool application under the Tools pull-down menu on the main NetView for AIX menu bar, but you can put it wherever you choose in the menu structure. Please refer to the *NetView for AIX Application Interface Style Guide* for suggestions about adding to the NetView for AIX menu structure. For more information about adding applications to the tools window, please see "Adding Your Application to the Tools Window" on page 34.

Examples of Tool Applications

Consider a tool application that checks the disk utilization of systems on a network. The application attempts to find all systems that have less than 10% disk space free. (The threshold might be configurable through a window provided by the application.) The application would be integrated into the NetView for AIX main menu bar. When selected by the user, the application will search for those systems whose free disk space is dangerously low. Once the tool application locates those systems, it will graphically accent, or highlight, those objects on the map. (Note that highlighting the symbol for an object does not alter the object database.) The user could then select one of those objects and telnet to that system to correct the disk usage problems. The application could provide help by placing text files in the appropriate directory, as described in Chapter 5, "Designing Application Help" on page 49.

This tool application relies on a number of NetView for AIX elements. It uses the application registration file to integrate into the NetView for AIX menu bar, and it employs the NetView for AIX Help system as well. This application uses the NetView for AIX EUI API to highlight the symbols that correspond to the computer nodes that are low on disk space.

Chapters about Tool Applications

Tool application developers should read a few key chapters in this manual. They include:

- Chapter 3, "Creating and Using the Application Registration File" on page 27
- Chapter 4, "Integrating Your Application with NetView for AIX Security Services" on page 43
- Chapter 5, "Designing Application Help" on page 49
- Chapter 6, "Understanding the NetView for AIX User Interface" on page 61 , and possibly other chapters in Part 2, "Working with the NetView for AIX User Interface"
- Chapter 11, "Understanding the NetView for AIX Management Environment" on page 167 , and possibly other chapters in Part 3, "Using the NetView for AIX Management APIs"

Map Applications

Map applications are applications that modify the contents of maps, that is, they update the NetView for AIX object database. To do this, map applications create and modify objects that represent an entity in a real-world network or system management domain. The map application assigns meaning to objects and the relationships between objects. Map applications can then create submaps that represent a subset of the map objects and their relationships. The next step, displaying the submaps to the user, is not handled directly by the map application. The NetView for AIX graphical interface acts as a mediator between the user and the map application, and displays submaps when requested by the user or by another application. The map application creates the underlying submaps, objects, and symbols that are presented to the user through submaps.

Examples of Map Applications

The IP Map application provided with the NetView for AIX program is an example of a map application. IP Map creates submap hierarchies that represent the different levels of network elements (networks, segments, nodes, and so forth) in an IP network. IP Map then populates each submap with icon and connection symbols that represent the objects on the network. Users can view a submap by double-clicking on a symbol or by selecting the submap from the submap list box available from the NetView for AIX main menu bar. The map application constructs the submaps, and the NetView for AIX graphical interface displays the submaps to the user.

Chapters about Map Applications

If you are writing a map application, you can expect to read portions from many of the remaining chapters in this manual. Map application developers need to use a wide variety of the functionality provided in the NetView for AIX developer's kit. Map application developers need to be familiar with such broad topics as:

- Chapter 3, “Creating and Using the Application Registration File” on page 27
- Chapter 4, “Integrating Your Application with NetView for AIX Security Services” on page 43
- Chapter 5, “Designing Application Help” on page 49
- Chapter 6, “Understanding the NetView for AIX User Interface” on page 61
- Chapter 10, “Map Events and Map Editing” on page 145 , and possibly other chapters in Part 2, “Working with the NetView for AIX User Interface”

Depending on the map application, you may need to refer to some or all of the chapters listed above.

Developing Applications for Control Desk Windows

The NetView for AIX graphical interface provides a special type of window called a *control desk* window. These windows give you more direct control of your application's execution than regular NetView for AIX windows allow. Applications designed to run in a control desk window require some special coding, which is described in Chapter 6, “Understanding the NetView for AIX User Interface” on page 61.

Most applications will fall into one of the application types described in this chapter. There may, however, be some blending of application types. For example, an application might possess the characteristics of both tool and map applications. In that case, you would need to read relevant sections from the appropriate chapters.

Integrating Your Application with Logging and Tracing

The NetView for AIX logging and tracing API, OVuTL, allows you to record information during the operation of your application. This information can be useful when you are developing and testing your application or diagnosing network management problems. Logging and tracing are the two tools provided for obtaining this kind of information. Logging and tracing have different objectives:

Logging

Logging is generally provided for the benefit of system or network administrators, developers, and some sophisticated end users. The messages you log should be understandable to this class of user. The system administrator uses the nettl process to configure, filter, and format log messages. The NetView for AIX logging process has practically no impact on the performance of your manager or agent, so logging can safely be turned on at any time.

Tracing

Tracing is far more comprehensive in its intent than logging. Tracing provides unambiguous evidence about the state of execution at key points in the code. Tracing is intended primarily for the developer, to assist during the testing phase of development. It is not intended for end-customer use; the information in the tracing files is typically understood only by the developer. When you turn tracing on, you should expect to experience a moderate degradation of performance. Therefore, tracing is generally on only during testing and debugging. The NetView for AIX program provides the OVuTL API to let you integrate your application with the common logging and tracing facility called nettl. This facility uses daemon processes to receive log and trace data from network applications, and to direct that data to its appropriate destination. For more information about the nettl subsystem, see the **nettl** man page, as well as other man pages to which it refers.

Table 1 shows the three routines in the OVuTL API:

Table 1. Functions in the OVuTL API

Routine	**Description**
OVuTLInit()	Initializes the software and hostname information for the logging and tracing output. You must call OVuTLInit() once only, before any calls to OVuLog() or OVuTrace().
OVuLog()	Enters a log message in the log file. By default this is /usr/OV/nettl.LOG00.
OVuTrace()	Enters a trace message in the trace file. By default this is /usr/OV/nettl.TRC0.

For more information about the OVuTL API, see the OVuTL() man page; it contains important details and examples of how to use this API.

Process Management

When you write an agent or daemon application for the NetView for AIX environment, you can choose how fully to allow it to cooperate with the NetView for AIX process management daemon, ovspmd. Process management controls the startup and shutdown of agent processes. It also sends information about agents' activities to nettl, the NetView for AIX tracing and logging subsystem, for potential use in problem determination.

From the perspective of a system administrator, process management is largely invisible. Two commands, **ovstart** and **ovstop**, execute the startup and shutdown functions, and one command, **ovstatus**, provides status information while the NetView for AIX program is running. However, as a programmer designing an application, you need to understand the different levels of control available to an agent application, and select the one that works best. The NetView for AIX program provides an application programming interface, the OVsPMD API, that helps you incorporate process management functions in your agent.

You can write one of three kinds of agents:

Well-behaved Agent
Uses the OVsPMD API to send status information regarding successful and unsuccessful initialization, and normal and abnormal termination. It also exits when it receives the command **OVS_CMD_EXIT** from ovspmd.

If you are writing a new program, you should create a well-behaved agent.

Non-Well-Behaved Agent
Does not use the OVsPMD API, and does not go into the background on its own. During shutdown, the ovspmd daemon sends **SIGTERM** to non-well-behaved agents to notify them of the need to terminate. Non-well-behaved processes that fail to terminate are sent **SIGKILL.**

If you have an existing agent that does not go into the background, you can decide to not modify it, and simply declare it a non-well-behaved agent.

Daemon Agent
Goes into the background on its own. While the ovspmd daemon can start such an agent, it cannot communicate with it thereafter, which means it cannot obtain status information about it or terminate it.

If you have an existing agent that does go into the background, you can decide not to modify it, and simply declare it a daemon agent. However, this results in poor integration of the agent and inefficient performance.

To create a well-behaved agent, use the OVsPMD API, which has four functions:

Table 2. Functions in the OVsPMD API

Function	Description
OVsInit()	Indicates that the agent is beginning its initialization phase. Returns a socket for communicating with ovspmd.
OVsInitComplete()	Used to indicate that the agent has finished its initialization phase.
OVsReceive()	Used to receive a command from ovspmd. Currently, the only command is OVS_CMD_EXIT, which indicates that your process should terminate.
OVsDone()	Used to inform ovspmd that the agent is terminating normally. One parameter is a text message used to indicate the reason for termination.

See the man page for OVsPMD_API() for additional details. In general, to create a well-behaved agent you should follow these rules:

1. Call the OVsInit() routine when your agent begins initialization. This gives you a socket for later communication with the ovspmd process.
2. After initialization is complete, and regardless of whether it was successful or not, call the OVsInitComplete() routine. A parameter to this routine indicates the initialization status; if initialization failed, your agent should exit.
3. Your agent should be organized around a select() loop, waiting for input from managers or from the managed object. You should select for reading on the file descriptor returned by the OVsInit() routine.
4. When select() indicates the file descriptor is readable, use the OVsReceive() routine to get the command from the ovspmd daemon. Currently the only command is OVS_CMD_EXIT, which means your agent should clean up, call OVsDone(), and exit.
5. If your agent exits on its own initiative (that is, without instructions from ovspmd), call the OVsDone() routine, indicating in the message parameter the reason for termination. This message will be logged by ovspmd along with other exit information.
6. Never go into the background (fork and exit in the parent); the child process cannot be managed by ovspmd.

The Local Registration File

You can create a local registration file (LRF) for any new agent that you create or install. The LRF allows managers to access your agent or application through the NetView for AIX program. You can also create an LRF for any new daemon you create. If you create an LRF for your daemon, you can integrate it with NetView for AIX process management.

The LRF is a specially formatted ASCII file that serves two purposes:

- The LRF contains information about the agent, including:
  - The agent's name
  - The full path name of the agent's executable code
  - How to start the agent
- The LRF contains information about objects managed by the agent.

Structure of the Local Registration File

Each line in the LRF has a specific purpose, as explained in Table 3.

Table 3. Purpose of Lines in the Local Registration File

Line	**Description**
First	Specifies the agent name and the pathname of its executable file
Second	Specifies process management information
Third (and subsequent)	Specifies objects managed by the agent, including their object classes, object instances, operations agents can perform on them, their protocol stack, and password

Note: The LRF must contain at least the first two lines of information. These first two lines describe the agent and are used for process management.

The third and any subsequent lines in the LRF pertain to any objects managed by the agent and are used by the object registration service. See "Third Line of the Local Registration File" on page 22 for more information.

General Syntax Notes for the Local Registration File

Each line in an LRF contains two or more fields. Each field, including the last field, is terminated by a colon. Some fields are optional, but you must include the colon terminator for the missing field.

The pound sign (#) indicates the beginning of a comment, which continues to the end of the line. Blank spaces are not permitted within any field, nor are multibyte characters permitted. Only printable ASCII characters are permitted in the first two lines of the LRF. The colon (:), comma (,), backslash (\), and pound sign (#) are permitted only as terminator characters.

First Line of the Local Registration File

The first line of the LRF specifies the name and location of the agent, as follows:

Name
A character string for the name of the agent being registered. You must ensure that the name is unique. No default value is given; you must supply a name.

This name is the name used when invoking the **ovstart, ovstop,** and **ovstatus** commands.

Path A character string that specifies the location of the agent executable code on disk. No default value given; you must supply the full (absolute) pathname.

Here is an example of the first line of an LRF:

```
ip_manager:/usr/OV/bin/MEGA/ip_mgr:
```

With this LRF, you could start the agent with the command **ovstart ip_manager**. This would invoke the program stored in /usr/OV/bin/MEGA/ip_mgr.

Second Line of the Local Registration File

The second line of the LRF defines the startup requirements for the agent.

There are five fields in the second line of the LRF. Each field is terminated by a colon (:). The field descriptions are:

Initial Start Flag An optional character string. The flag defaults to **OVs_NO_START**, indicating that the agent starts only if it is explicitly named on the ovstart command, or if another agent that depends on this one is started.

OVs_YES_START means the agent starts automatically when ovstart is run with no arguments.

OVs_NO_STOP means the agent is not stopped unless it is explicitly named on the ovstop command.

Dependencies A list of agent names, separated by commas, that must already be running before your agent can be started. Use the names as they appear in the *Name* field of the appropriate LRFs.

This field is optional; if nothing is specified, the agent can be started at any time.

Arguments This field contains a series of character strings, separated by commas. The arguments specified in this field are passed to the agent as it starts, just as though they were passed from a user at the command line.

This field is optional; if nothing is specified, it means that no arguments are required.

Behavior This optional character-string field specifies how the agent will interact with the ovspmd process management daemon. Agents can be well-behaved (OVs_WELL_BEHAVED), non-well-behaved (OV_NON_WELL_BEHAVED), or daemons (OVs_DAEMON). The default is **OVs_NON_WELL_BEHAVED**.

Timeout This integer field is optional. The default is 5 seconds.

For well-behaved agents, this field is used to manage evident startup failures, as follows:

- Your agent invokes OVsInitComplete() and returns OVS_RSP_FAILURE in the status code parameter but fails to terminate before the timeout expires.
- Your agent invokes OVsDone() but fails to terminate before the timeout expires.

In either case, the ovspmd daemon terminates the agent process, sending it **SIGTERM** and, if necessary, **SIGKILL**.

If your agent is either non-well-behaved or a daemon, the ovspmd daemon waits until the timeout expires before starting any agent that lists your agent in its *Dependencies* field. Also, after the ovspmd daemon sends the **SIGTERM** command to your agent, it waits until the timeout expires before sending it a **SIGKILL** signal.

Here is an example of the second line of an LRF:

```
OVs_YES_START:pmd,ems_sieve_manager:-v,-n:OVs_WELL_BEHAVED:15:
```

Third Line of the Local Registration File

The first two lines of the LRF contain information that relates to process management. This information is used by the ovspmd daemon when it sends start and stop requests to an agent or application.

The third line of the LRF contains information about the objects that are managed by the agent. This information is stored in the object registration service database, which is maintained by the orsd daemon. When a message from a manager or agent is sent to a specific object, the pmd daemon consults a directory of orsd database entries to determine the location of the object.

Table 4 describes the fields in the third line of the LRF. For more detailed information on these fields, consult the LRF man page.

Table 4. Fields in the Third Line of an LRF

Field	**Description**
Object Class	The location of the object class in the registration tree. There is no valid default value for this field. You can use a wildcard character (*) in this field only if the agent uses SNMP through the XMP interface.
Object Instance	The fully distinguished name (FDN) of the object instance, created by concatenating the Relative Distinguished Names (RDNs) in the containment hierarchy. There is no valid default for this field. This field is not used for agents that use SNMP through the XMP interface; instead, a hyphen is used as a placeholder. A wildcard character indicates that the agent manages more than one instance of the class; in particular, it manages all instances whose instance values match this field up to the point of the wildcard. A wildcard character can be used only as the last character in this field. A tilde (˜) is a special “target-host address” character, used mainly in the LRFs of agents. This character is replaced with the IP address of the target host.
Operation	Specifies the operations that the agent can perform on the managed object. If a wildcard is specified, it means all operations except EVENT_REPORT are supported.
Protocol Stack	Specifies the protocol stack that must be used to reach the object. If you leave this field blank, the default value is **OVs_ACMOT_TCP**.
Password	A password is necessary only for SNMP proxy resolution. If you leave this field blank, the default value is **null**.

Here is an example of the third line of an LRF:

```
1.3.6.1.2.1.4:1.3.6.1.2.1.9.3=˜:OVs_GETR:OVs_ACMOT_TCP:OVs_Pwd=MySecret:
```

Example of a Local Registration File

This example of a local registration file shows an IP agent that manages all instances of CMOT Internet IP objects that exist on the same node as the agent.

```
# IPAgnt Local Registration File OVs_CMOT_TCP
IPAgnt_A.017.0_MegaCorp_International:/usr/OV/bin/ip_mgr:
OVs_YES_START:pmd,ems_sieve_agent:-v,-n:OVs_WELL_BEHAVED:15:
1.3.6.1.2.1.4.20.1:1.3.6.1.2.1.9.3=˜/1.3.6.1.2.1.4.20.1.1=*:*:OVs_ACMOT_TCP:OVs_Pwd=Hush:
```

The first line in the preceding example is a comment line. The next three lines correspond to the first, second, and third lines in an LRF, as follows:

1. Name and location of agent
   - Name
   - Full pathname
2. Process management information for agent
   - Initial start flag
   - Dependencies
   - Arguments
   - Behavior
   - Timeout
3. Object description
   - Object class
   - Object instance
   - Operations that agent can perform on the managed object
   - Protocol stack used to reach the object
   - Password (used only for SNMP proxy resolution)

Rules for Building a Local Registration File

There are some rules you should follow when you build an LRF:

- The first line contains global information and is not specific to any object it manages. There is only one such line in an LRF.
- The second line contains startup information. If any of the fields in this line are not specified, the terminator (:) associated with that field must still be present, because all fields in the file are position-dependent.
- The object must be defined in either Internet MIB (version 1) or a subset of ASN.1 universal simple types.
- An instance of an object cannot be managed by more than one agent.
- A wildcard is allowed in an object class only if the object instance is an SNMP variable, as indicated by a hyphen (-).
- When an asterisk wildcard is used in an object instance field, it is allowed only at the end of the sequence of RDNs.
- The use of a wildcard in an operation field represents all operations except EVENT-REPORT.
- Ensure that use of wildcards in an LRF will not result in a duplicate entry in another LRF, since all LRFs are consolidated.
- Start a new entry on a new line.
- Entries cannot contain duplicate object class, instance, and operations.

- Blanks are not permitted in any field.
- All fields must be specified. If a default value is used, the accompanying field terminator (:) must be used.
- The distinguished name consists of one or more RDNs. Only the distinguishing attribute is used for routing.

There are many ways to integrate your application with the NetView for AIX program. You must choose the method that fits your needs best. One task which must be completed for any application is creating an application registration file, which is the subject of the next chapter.

The $LANG Environment Variable

The $LANG environment variable is used to indicate the language in which information is written. The $LANG variable is often used as a directory name in the path names of user-configurable files. For example, the path name for application registration files is /usr/OV/registration/$LANG.

For English-language versions of the NetView for AIX product, C is always used in path names, even if the $LANG variable is set to En_Us. When you create new registration files, bitmap files, help files, or other files whose path name contains $LANG as the name of a directory, be sure to specify C as that directory name. Thus, any new application registration files should be placed under the /usr/OV/registration/C directory.

Customizing NetView for AIX Startup

Some customers and application vendors want to set environment variables or execute scripts when the nv6000 command is executed. These modifications should not be made in the /usr/OV/bin/nv6000 script itself, or in /etc/netnmrc, because these files are subject to modification with any service update or new version of the NetView for AIX program. User or vendor modifications to the nv6000 script and the netnmrc file will not be preserved when migrating to a new version of NetView for AIX, or when applying a service update that affects those files.

To enable customers and vendors to make modifications that will not be lost by upgrading, the nv6000 startup script runs the script named /usr/OV/bin/applsetup (if it exists) just prior to starting the user interface. This script is run in the same process as the nv6000 command and thus allows the setting or changing of environment variables and other custom actions to be performed just as though the code had been edited into /usr/OV/bin/nv6000 itself.

You can edit /usr/OV/bin/applsetup to add individual commands or commands to run other shell scripts. Each such command must run its script in the current process if that script sets or changes environment variables that are to be passed to the EUI at startup time. For example, the following command runs "myscript" in the current process:

```
. /usr/OV/bin/myscript
```

If you have made any modifications to the nv6000 script, we strongly recommend that you move them to /usr/OV/bin/applsetup to avoid possible loss of startup customization in the future.

For modifications that start processes that you want to have running independent of the NetView for AIX user interface, and that require root access, you can use the new /usr/OV/bin/netnmrc.aux script. This script is run at the end of netnmrc. Entries in this script do not have to run in the current process. If you have made any modifications to /etc/netnmrc, we strongly recommend that you move them to netnmrc.aux to avoid possible loss of startup customization in the future.

Since the applsetup and netnmrc.aux scripts reside in /usr/OV/bin, they will automatically be backed up and migrated so long as the /usr/OV/bin.USER category is selected for backup.

Chapter 3. Creating and Using the Application Registration File

This chapter describes how to create and use application registration files to integrate your application with the NetView for AIX program. You must create an application registration file (ARF) for each application that you write. The ARF is very important in defining how your application interacts with the NetView for AIX program.

A complete syntactical description of the ARF and other registration files is provided in the OVwRegIntro man page. For convenience, all registration file entries use a syntax that resembles the C programming language.

Understanding Application Registration Files

Application Registration Files are stored in the directory **/usr/OV/registration/$LANG**. The *$LANG environment variable* provides support for native language support. If not otherwise defined, *$LANG* is assumed to be C, which means no native-language support is provided.

The registration files provide a static mechanism to configure NetView for AIX behavior. Registration files are complemented by the NetView for AIX Application Programming Interfaces (APIs). When used in conjunction with the registration files, the APIs provide the additional support required to make your application an integral part of the NetView for AIX program. The NetView for AIX APIs are described in Part 2, "Working with the NetView for AIX User Interface" on page 57 and Part 3, "Using the NetView for AIX Management APIs" on page 161.

Note: The following examples use quotation marks around character strings coded in the ARF. Quotation marks are required on the Command entry and on the names of menu items in a Menu block. Quotation marks are required on other character strings if they contain any blank characters or if the string contains a keyword, such as **application**. Otherwise, quotation marks are optional.

Creating the Application Block

Each application is integrated into the NetView for AIX program using a structure in its ARF called an application block. The application block defines many aspects of the interface between the application and the NetView for AIX program. Entries and sub-blocks in the application block specify the following behavior:

- Menu bar structure
- Instructions for invoking the application command
- The version, description, and copyright strings displayed for the application
- Where application help information is located
- The format of dialog boxes that the NetView for AIX graphical interface can display on behalf of the application

An ARF contains a single Application block that represents a single application. An application can be defined only once; its name must be unique. The Application block has the following syntax:

```
Application "<application name>" {
    ..
    one or more application specifications
    ..
}
```

The <application name> is replaced by the string by which the NetView for AIX program will refer to your application. The remainder of this chapter describes application specifications in more detail.

Specifying the Application's Version, Copyright, and Description

Users can view an application's version string, description, and copyright by selecting the Applications Index from the main NetView for AIX Help menu. For this information to be displayed properly, it must be specified in the ARF. The version is specified as a string in a Version entry. The application's purpose is specified in a Description block. The copyright is defined in a Copyright block. The following example demonstrates the use of these blocks:

```
Application "My app" {
    ..
    Version "1.01";
    Description {
        "Sample application description",
        "goes here."
    }
    Copyright {
        "(C) 1995 International Business Machines Corp."
    }
    ..
}
```

The Version text string must be a single line. The Description and Copyright blocks are composed of one or more text strings separated by commas.

Defining the Help Directory

All applications integrated under the NetView for AIX program can provide online help information. The HelpDirectory entry in the application block defines where the help information is located.

The HelpDirectory section in the ARF specifies the directory containing the application's help files. The NetView for AIX program assumes the string **/usr/OV/help/$LANG/** is prefixed to the directory name specified in the HelpDirectory entry. Assume that the default language is C, and consider this example:

```
 Application "My app" {
    ..
    HelpDirectory  "thisapp";
    ..
}
```

In this case, the help files are found in the /usr/OV/help/C/thisapp directory.

Instructions for defining the actual application help files are not presented here. Chapter 5, “Designing Application Help” on page 49 describes how to implement help files for your application. Refer to the *NetView for AIX Application Interface Style Guide* for guidelines about designing online help.

Integrating Applications into the Menu Structure

A key feature of the NetView for AIX program is its ability to integrate network and systems management applications with a common menu interface. This section shows you how to integrate applications into the NetView for AIX menu structure through the ARF.

The NetView for AIX program allows you to integrate your application either on the main menu bar or under a menu cascade. Regardless of where your application is integrated in the NetView for AIX menu structure, the following tasks are required to register your application:

Step 1. Specify the menu item on the main menu bar under which your application will be integrated.

Step 2. Specify the relationship, or linkage, between the main menu bar item and the next menu item or pull-down menu.

Step 3. Continue defining menu items in cascading menus as needed.

Step 4. When you have reached the menu entry in which your application should be invoked, define an Action block that specifies how to invoke the application.

Here are the steps in more detail.

Defining the Menu Structure

Application integration begins in the main menu bar. A MenuBar block specifies the menu label on the main menu bar under which the application will appear. You can specify an optional mnemonic character to enable the user to select the item by typing a single character. For example:

```
Application "My App" {
   ..
   MenuBar "Configure" _C {
        ..
   }
   ..
}
```

In the previous example, application integration begins under the Configure menu bar entry. The user can select this menu item either by using the mouse or by typing **C** once the menu bar is activated for keyboard input.

If the specified menu item does not exist, the NetView for AIX program creates it for you. If it already exists, the NetView for AIX program will make the association to the existing menu item.

The main menu item on the menu bar is now defined. The next step is to define additional menu items and the application invocation. These are defined by a

Menu statement within the MenuBar block. The Menu statement specifies the menu label and its function. One of the following three functions can be used:

f.menu Provides the declaration of a menu cascade within a menu. Menus in the NetView for AIX menu structure will be extended to include the new entry, or if one does not exist, a new cascading menu will be created.

f.action Associates an application invocation, or action, with the menu item. When the user selects the menu item, the application is invoked appropriately. The application is notified of the selected action.

! Provides a short-cut mechanism to integrate shell commands into menu selections. Applications invoked in this way are not notified of the user's selected action, and selection rules (described in “Defining Application Invocation” on page 36) cannot be used. If you want these features, use the f.action function instead.

Using the Menu Structure

The following example demonstrates these menu concepts. The details of invoking the application will be described in the next section.

```
Application "My App" {
    ..
    MenuBar "Monitor" _M {
        "Local Network" _L f.menu NetItem;
    }

    Menu NetItem {
        "Trends..."      _T f.action Trends;
        "Statistics..." _S f.action Stats;
    }
    ..
}
```

In this example, the user can select the **Monitor** menu label from the main menu bar. A pull-down menu containing the **Local Network** menu item is displayed. Selecting the **Local Network** menu item causes a cascading menu to appear that contains two more choices, **Trends...** and **Statistics....** These choices are specified in the Menu block, which is associated with the **Local Network** item by specifying the name NetItem, which appeared in the f.menu specification for that menu item. Selecting either of these menu items causes the appropriate applications to be invoked.

Notice the single-character mnemonics in the example menu statements. Mnemonics are one of the following optional modifiers for a menu statement:

- A single-character mnemonic for keyboard selection, as in the example.
- A precedence value indicating how the menu item should be ordered. Menu items are displayed according to the ordered precedence, and, within items of equal precedence, in the application name order. Precedence is specified as an unsigned integer value; 100 is the highest precedence, and zero is the lowest.
- A keystroke sequence that invokes the menu selection without displaying the menu. This is also called a keyboard accelerator and is specified using the Motif* keyboard binding syntax.

The syntax for these options is specified in the OVwRegIntro man page.

Note: Consult the *NetView for AIX Application Interface Style Guide* for guidance when adding your application into the NetView for AIX menu structure. In particular, avoid creating new items on the main menu bar unless absolutely necessary. Reuse existing menu structures wherever possible, and do not create menu cascades more than three levels deep.

Using the Command Entry

The Command entry lists the complete command line used to invoke the application. Whether the user selects the application through a menu item or through an executable map symbol, this command will be used to invoke the program. Commands that are tty-based (that is, non-X Window System applications) must be run in an X-terminal window (**aixterm(1)**, for example). The following example illustrates the Command entry:

```
Application "editor" {
    ..
    Command "aixterm -e /usr/bin/vi";
    ..
}
```

Quotation marks are required around the command string, even if it contains no blanks.

Because it is defined at the application block level, the command is considered global. The application command will be invoked the same way every time, even if the user selects the application in different ways. Commands can be defined on an individual, per-action basis. Applications can be invoked in different ways (for example, with different arguments) depending on the user's selection. See “Defining Application Invocation” on page 36 for more information on per-action commands.

Using Shell Environment Variables

The NetView for AIX program provides a run-time environment for applications, allowing applications to access values of NetView for AIX environment variables. Applications have access to a number of key items, including the menu items used to invoke the application and the object selection list. These environment variables can appear on the command line that invokes the application. The following NetView for AIX environment variables are available for your use:

OVwSelections
A string containing the selection names of the objects in the selection list when the application is invoked. The names are separated by blank characters.

OVwNumSelections
The number of selection names in the selection list when the application is invoked.

OVwSelectionn (n=1..10)
OVwSelection1 is set to the selection name of the first object in the selection list, OVwSelection2 is set to the second name, and so on. These strings are returned in the order in which their associated objects were selected. The selection list is limited to ten elements.

OVwActionID
The name of the action by which the application is invoked. Actions are defined in the ARF and are described later.

OVwMenuItem
If the application is invoked through a menu item, this environment variable will contain the label of the menu item that caused the action. If the application is invoked from the Tools window, this variable will contain the tool's label as specified in the Tool block for that application.

This example demonstrates a command that is invoked with the first element in the selection list:

```
    Application "Telnet" {
        ..
        Command "/usr/bin/telnet $OVwSelection1";
        ..
    }
```

If you use selection names in your Command entry, you must understand the format of the selection list. If the selected string contains a space, it will be divided at the space, and argv(1) will contain only the string preceding the space. You can handle this situation and use the entire string by placing \" before and after the selection variable in your Command entry. The following example illustrates this usage:

```
    Application "Myprog" {
        ..
        Command "/usr/bin/myprog \"$OVwSelection1\"";
        ..
    }
```

Use this technique carefully to avoid unexpected results. For example, if you use \" around a list of selection variables, they will be collated into one string rather than being treated individually.

Note: When an application is invoked from an object menu, the selection list consists of the object from whose menu the application was selected. Any previous object selections made by the user are ignored. For information about adding your application to an object menu, please see “Adding Your Application to an Object Menu” on page 34.

Using Process Flags to Control Execution

The NetView for AIX program can manage your application in a number of ways. By default, the NetView for AIX program invokes your application at run-time when the user selects the appropriate menu item or executable symbol. Each selection by the user causes another instance of the application to be loaded and executed. You can change this behavior by adding special flags to your command entry in the application block. For example,

```
    Application "My app" {
        ..
        Command -Initial -Shared -Restart
"/usr/local/bin/your_app";
        ..
    }
```

There are three flags defined: *Initial, Shared,* and *Restart*. The following list describes their use:

Initial Tells the NetView for AIX program to start the application when the NetView for AIX program is started. The application will be invoked by the Command line in the application block.

Shared Tells the NetView for AIX program that only a single instance of this application command should be running at any time. This application instance is shared, and will handle action requests that occur after it has started.

Restart Tells the NetView for AIX program to restart the application if it should ever exit. This flag is intended for applications that are required for normal NetView for AIX operation, such as those managing the semantics of NetView for AIX maps.

If you want to use more than one process management option, combine them in the same Command entry as shown in the example.

Note: If your application serves one purpose and always performs the same function when called, an ARF specification will be sufficient. However, if your application is shared and performs several different functions, it must call the NetView for AIX API routine OVwAddActionCallback(), which is described in “Defining Callback Routines for Events” on page 63.

Defining a NameField

A single NetView for AIX object can have several names, such as:

- A selection name (NetView for AIX's unique name for the object)
- An IP hostname (for IP networks)
- A user-defined or application-defined name

When a user selects an object on a map, the NetView for AIX selection list is built using the object's selection name by default. The NameField section lets you choose other name forms that are used to construct the selection list. You can, for example, request that only IP Hostnames are used in the selection list. Using a NameField entry guarantees that actions or commands are allowed only if all objects in the selection list have that name field. Otherwise, the menu item is greyed out.

You can use any of the preregistered NetView for AIX name fields in the NameField section, or you can create your own. For instance, a previous example demonstrated how to invoke telnet with an NetView for AIX shell environment variable. This works well if the selection item is the same as the host name and the host is capable of supporting telnet. However, it is be inappropriate for a modem. By supplementing the Command section of the registration file with a NameField section, you can guarantee that the command is invoked with a list of selected items appropriately filtered. You can extend the telnet example by filtering out all selected items that do not have host names, as shown in the following example:

```
Application "Telnet" {
    ..
    Command "/usr/bin/telnet $OVwSelection1";
    NameField "IP Hostname";
    ..
}
```

In this example, the command will be enabled only if all objects in the selection list have an IP Hostname name field. The command will be invoked using the first object in the selection list as an argument.

You can supply several name types, separated by commas, in the NameField entry. The NetView for AIX program will check the object's names against each name specified in the NameField statement. If several names are specified in the NameField statement, the NetView for AIX program will use the first name that is valid for the object. The NameField entry is a convenient way to filter selections from the selection list. You can construct more sophisticated filters using selection rules, which are specified in the Action block.

Adding Your Application to an Object Menu

In addition to the menu structure accessible through the main menu bar, object-related menus can be invoked by pointing to an object on the submap and pressing and holding the third mouse button. Refer to the *NetView for AIX User's Guide for Beginners* for information on using these menus. If your application is one that operates on an object, you can add your application to the menu that is displayed in this way. To add your application to the object menu, code an ObjectMenu block. The syntax of the ObjectMenu block is similar to that of the Menu block, except that no name is specified:

```
Application "My App" {
    ..
    MenuBar "Monitor" _M {
        "Local Network" _L f.menu NetItem;
    }

    Menu NetItem {
        "Trends..."      _T f.action Trends;
        "Statistics..." _S f.action Stats;
    }
    ObjectMenu {
        "Trends..."      _T f.action Trends;
        "Statistics..." _S f.action Stats;
    }
    ..
}
```

Your ObjectMenu entries can invoke the same Action block specification that is used for the main menu entries, as in the example above, or they can invoke different actions that are defined in another Action block.

Note: When an application is invoked from an object menu, the selection list consists of the object from whose menu the application was selected. Any previous object selections made by the user are ignored.

Adding Your Application to the Tools Window

As part of its graphical user interface, the NetView for AIX program provides a tools window which gives users easy access to commonly-used applications in a drag-and-drop style. You can add your application to the tools window by coding a Tool block in your ARF. The Tool block has the following syntax:

```
Tool <Precedence> "Label"   {
   Icon  [Bitmap "Filename"] | [Gif "Filename"] | [Solid "Color"];
   LabelColor "Color";
   DragBitmap "Filename";
   SelectionMechanism double-click, drag-drop;
   Action "Action Name";
}
```

The Tool block entries are described in the following list:

Precedence An indication of the importance of the item on the Tools Window. Tools are listed in the Tool Window according to precedence. Items of the same precedence are ordered according to the order in which they were registered. The precedence is specified as an integer from 0 to 100 inclusive, enclosed in brackets (for example, <50>). An item with a precedence of 100 is listed at the top of the Tools Window. Note that the Control Desk will always be at the top of the Tools Window, even if there are other tools registered with a precedence of 100.

Label The label to appear beneath the icon in the tools window. This label will appear in the OVwMenuItem environment variable when the application is invoked from the Tools window.

Icon Specify either Bitmap or Gif with the fully-qualified name of the file containing a bitmap or gif file for the tool icon, or Solid with the solid color for the icon.

DragBitmap The fully-qualified name of the file containing a bitmap to be used for the cursor while dragging the icon. Use only if SelectionMechanism is drag-drop. This bitmap may contain only two colors. The drag bitmap requires two files, with suffixes of .p and .m attached to the file name specified in the Tool block. These files are similar to those required to build a new symbol, and are described in "Creating Bitmaps for NetView for AIX Symbols" on page 105.

SelectionMechanism
The user action (double-clicking the mouse button or drag-drop) used to invoke the application. You may code either value or both separated by commas.

Action The name of an Action block that specifies the command to be invoked when this application is selected.

In the following example of a Tool block, the tool is represented on the tools window by a Gif icon, and is selected by double-clicking on the icon:

```
Tool <50> "Graph Demo" {
   Icon Gif  "/usr/OV/gif/C/graph.gif";
   LabelColor  "black";
   DragBitmap  "/usr/OV/bitmaps/C/graph";
   SelectionMechanism drag-drop;
   Action      "graphDemo";
}

Action "graphDemo" {
    /*  definition of graphDemo action  */
}
```

This tool icon has a precedence value of 50. It will be placed closer to the top of the Tools Window than other applications with a lower precedence value. If there are other icons with a precedence value of 50, the icons will be arranged alphabetically by application name.

Note: If your application is to be dropped into a control desk window, you must do some extra work. See "Developing Applications for Control Desk Windows" on page 74 for details.

Defining Application Invocation

Once the menu structure is in place, the last step is to define how the application is invoked. Application invocation is defined in an Action block. The Action block specifies how the application command is invoked, how the process is managed, and how the application uses the selection list from the map. The following definitions can be used in an Action block:

SelectionRule
Provides a mechanism to guarantee that the application is invoked only if the selected objects meet predefined criteria. The selection rule is a logical expression involving capability fields. The logical expression can use logical AND (&&), logical OR (||), or logical NOT (!) operators.

MaxSelected
Specifies the maximum number of objects that can be selected on the map for the action to be enabled. If this value is not specified, any number of objects may be selected for the action. If this value is set to zero, no objects may be selected.

MinSelected
Specifies the minimum number of objects that must be selected on the map for the action to be enabled. If MinSelected is set to some value, the selection list must contain at least that number of objects. If MinSelected is not set, any number of objects can be selected. If MinSelected is not set and a SelectionRule is set, the selection list must contain at least one object. If MinSelected is set to zero and a SelectionRule is set, the selection list can contain zero or more objects.

Security Specifies that this action is available only if NetView for AIX security is active. Chapter 4, "Integrating Your Application with NetView for AIX Security Services" on page 43 has more information about NetView for AIX security.

Command
Specifies the command line used to invoke the application on a per-action basis. Commands specified in an Action block override commands specified globally in the Application block. If no command is specified globally, a command must be specified for each action defined.

Process Flags
You may define specific process management instructions on a per-action basis. The *Initial, Shared,* and *Restart* flags may be used on a per-application basis and override any global process flags.

NameField
The NameField entry provides the same object name field validation as described in the Application section, but on a per-action basis.

CallbackArgs
Using the NetView for AIX EUI API, you can designate a callback routine that is invoked when the user selects this action. Using the CallbackArgs entry, you can specify a string containing arguments that are passed to the callback routine. Callback routines are described in "Defining Callback Routines for Events" on page 63.

The following example illustrates an action block specification:

```
Application "My App"
    ..
    MenuBar "Monitor" _M {
        "Network Activity" _A f.menu "Net Activity";
    }
    Menu "Net Activity" {
        "Trends..."     _T f.action Trends;
        "Statistics..." _S f.action Stats;
    }
    Action Trends {
        SelectionRule isNode && isDevice;
        MinSelected 1;
        MaxSelected 1;
        Security;
        Command "/usr/local/bin/myapp $OVwSelection1";
        CallbackArgs "-trends -verbose";
    }
    ..
}
```

In this example, if NetView for AIX security is active, the user can select the **Trends...** menu item from the menu cascades to request that the application be invoked. The NetView for AIX program executes the application only if there is a single object selected on the map, and that object has the isNode and isDevice capabilities. If these conditions are met, the application is invoked with the name of the selected item. The CallbackArgs statement defines a string that is passed to the application's callback routine when the user invokes this action.

Using Dialog Boxes

The NetView for AIX graphical user interface can be used to perform a variety of map operations that might affect your application. These map operations can result in user interaction through a NetView for AIX dialog box. If your application supports any of the four basic map operations (Add, Describe, Connect, or Configure), you can let the NetView for AIX graphical user interface construct dialog boxes for you. This section explains how to use this automatic dialog box generation.

The definition of fields to be presented in dialog boxes is called field enrollment, because each field presented in a dialog box is a field in the NetView for AIX object database. You can define the basic characteristics of dialog boxes and leave the details of dialog box construction to the NetView for AIX program. These dialog boxes are used only when the user selects the Add, Describe, Connect, or Configuration operations. Limited control over dialog box layout is provided.

Note: If you have particular dialog box needs that are not met by NetView for AIX's field enrollment capability, use OSF/Motif* to implement your own dialog box directly.

Defining Dialog Boxes with the Enroll Block

Dialog boxes are defined using Enroll blocks. A separate Enroll block is used for each dialog box type that you define. The Enroll block has the syntax:

```
Application "My app" {
    ..
    Enroll <dialog type> {
        one or more field enrollment specifications
    }
    ..
}
```

The <dialog type> is one of the following four map operations:

Add	The dialog box is presented when a user adds an object to the map, using the `Edit..Add Object` menu function.
Describe	The dialog box is presented when a user invokes the describe action on an object, using the `Edit..Modify/Describe` menu function.
Connect	The dialog box is presented when a user attempts to connect two objects on the map, using the `Edit..Add Connection` menu function.
Configuration	The dialog box is presented when a user requests to change per-map configuration parameters for an application, using the `Edit..Modify/Describe` or `File..Describe Map` menu function. This dialog box is application-specific and is not related to an object.

The Enroll block defines which of the four map operation dialog boxes your application supports. The next steps are to:

1. Define logical expressions within the Enroll block that let you display different fields, which depend on the capabilities of the selected objects. These expressions are called rules.
2. Define field specifications for each field within a rule. The field specifications define the label, display policy, and editing policy for each field.

Defining Rules within Enroll Blocks

Rules are logical expressions that use object capabilities to control when fields are displayed. Rules have the following format:

```
Enroll <dialog type> {
   if <capability expression> {
      <field specifications>
   }
   ...
}
```

In its simplest form, the logical expression is an object capability. Object capability fields are described in “Field Flags” on page 81. You can use logical AND (&&), logical OR (||), or logical NOT (!) operators to construct expressions. For example, consider a Describe dialog box. You may want to display one set of fields if the user selects an object with network interface capabilities, and another set of fields if the user selects an object with either node or router capabilities. The following ARF segment illustrates this case:

```
Application "My app" {
    ..
    Enroll Describe {
        if isNetworkInterface {
            Field "IP Address" {
               ...
            }
        }

        if isNode || isRouter {
            Field "IP Hostname" {
               ...
            }
        }
    }
    ..
}
```

In the previous example, the IP Address field will be displayed if the selected symbol has the isNetworkInterface capability, while the IP Hostname field will be displayed if the selected symbol has isNode or isRouter capabilities.

Note that rules are based on object capabilities. Rules in the Add and Describe dialog types operate on a single object. However, the Connect dialog type involves two end points, or objects, to be connected. In this case, two logical expressions, separated by a comma, are required, as shown in the next example:

```
Enroll Connect {
   if ( isNode || isRouter ), isRouter {
      <field specifications>
   }
}
```

In this case, the first end point must be an object that has either isNode or isRouter capabilities, while the other end point must have the isRouter capability. Selection order is not important.

The Configuration dialog type is not related to an object and therefore does not require rules. The following example illustrates a Configuration Enroll block:

```
Application "My App" {

 Command -Shared "$HOME/myapp";
 ...
 Enroll Configuration {

   Field "myapp_string" {
     Label "My string field: ";
     EditPolicy Edit;
     DefaultValue "No Comment";
     }
          ...
   Field "myapp_enum" {
     Label "My enumeration field: ";
     EditPolicy Edit;
     DefaultValue "Mode 1";
     }
 }
```

This example also shows two field definitions, which are described in the next section.

Using Fields in the Enroll Block

Field specifications within an Enroll block define which object database fields are presented to the user, as well as their display format. A field specification can contain the following information:

Field	The name of the field in the object database (required)
Label	An optional string used as field label in the dialog box. By default, the label is the same as the field name.
EditPolicy	An optional editing policy for the field. By default, fields can always be edited. You may specify that the field is read-only (NoEdit) or that the field may be edited only when it is used in a new map or new object (EditOnCreation).
IntegerDisplayPolicy	An optional display policy for integer fields. By default, integers are displayed as 32-bit signed decimal values. You may also specify Unsigned, Hex, or Octal, or IPAddr for conventional IP Address dot notation format.

The following example illustrates how to specify fields within an Enroll block. In this example, the field contains an IP address. This means:

- The field should be displayed only if the symbol is a node. This is accomplished with the `if isNode` specification preceding the field definition.
- The field label should be IP Address:
- The field value should be displayed in IP address format.
- The user is prevented from changing the value.

```
Application "My map app" {
    ..
    Enroll Describe {
        if isNode {
            Field "IP Address" {
                Label "IP Address:";
                IntegerDisplayPolicy IPAddr;
                EditPolicy NoEdit;
            }
        }
       ..
    }
    ..
}
```

Many Enroll blocks contain multiple field specifications. You can examine the application registration files provided with the NetView for AIX program for more examples. The OVwRegIntro() man page contains a complete description of entries within Enroll blocks.

Providing Enroll Block Help

Each Enroll block defines the format for a dialog box generated by the NetView for AIX program. The NetView for AIX program provides a comprehensive help system to assist users in their tasks, and dialog-box help is part of this help system. The help information is displayed when the user selects the Help button in the dialog box. Use a HelpFile entry in your Enroll block to specify the location of a dialog-box help file. The HelpFile entry has the following format:

```
Application "My app" {
   ...
   Enroll Describe {
      if isNode {
         HelpFile "OVw/Dialogs/describe_help";
         Field "IP Address" {
            ...
         }
      }
   }
   ...
```

The help file follows a specific format that is described in the *NetView for AIX Application Interface Style Guide*. The path of the help file is specified in quotes and is relative to /usr/OV/help/$LANG/HelpDirectory/appname, where appname is the name given to your application in the Application block of its ARF.

Registering for Map Editing Events

After defining your dialog box with an Enroll block, if you have at least one “Command” statement defined in your registration file, you must register your application to be notified when the user performs actions that require dialog-box input. Your application must register both Query and Confirm callback routines. This process is described in “Participating in Map Changes” on page 152. If you have no “Command” statements defined in your registration file, your Enroll blocks are accepted by NetView for AIX and will be displayed whenever those actions are invoked by the user.

How Registration Files Are Processed

All application registration files (ARFs) are read each time the **nv6000** command is issued. Each time the NetView for AIX program is started, as part of its initialization process, it searches the registration-file directory, /usr/OV/registration/$LANG, and its subdirectories, for files. Any file found in this directory is assumed to be a registration file. For each ARF found, the NetView for AIX program executes the following steps:

Step 1. Opens the file.

Step 2. Parses the file for correctness.

Step 3. If the file is valid, the NetView for AIX program adds the item to the designated menu and defines actions and other ARF specifications.

Step 4. If the file contains errors, the NetView for AIX program prints error information to `stderr`, indicating the name of the file and the number of the line where the error was found. Errors can be caused by syntax errors or duplicate definitions.

Symbol Type Registration Files (STRFs), described in “Defining Symbols with Symbol Type Registration Files” on page 109, are processed in the same way as Application Registration Files. Field Registration Files (FRFs) are processed in the same way, but only when the nv6000 command is invoked with the `-fields` option. See “Using the Field Registration File” on page 80 for more information on FRFs.

Chapter 4. Integrating Your Application with NetView for AIX Security Services

Version 4 of NetView for AIX provides new distributed security services for network administrators to control the access users have to applications and platform functions, to audit usage, and control access to NetView for AIX itself. As an application developer, you can choose to make use of the security services by providing default security settings for your application.

To make use of NetView for AIX security services, you need to do two things:

- Provide a Security Registration File (SRF) to register sensitive resources.
- Code security APIs. This is only required if your application can be invoked outside of the EUI menu bar, or if it needs to use other security services (such as auditing).

Deciding Whether to Secure Your Application

Basically, the decision on whether to make your application secure depends on whether your application has any sensitive resources. Does your application have any of the following?

- EUI menu bar items
- Object context menu items
- Items on the Tool Window
- Commands that can be executed from the command line
- Resources that are configurable (with the new auditing capability, should the network administrator be notified of a configuration change in your application?)

If your application meets any of these criteria, you should define default security settings for your application.

If you choose not to provide a security framework around your application, the application can still be used in a network that is taking advantage of NetView for AIX security services, although your particular application will not be secure.

A Guide to Integrating

This section outlines the steps for integrating your application with NetView for AIX's distributed security services.

1. Determine which resources need to be secured, as outlined in the previous section.
2. Create a default Security Registration File (SRF) for the resource. The SRF is a seed file that specifies the access different security groups have to resources. You can secure resources at several different levels of granularity, from an entire application to class methods to individual menu items. Within the SRF, you define the name of the application, access permissions for the resource, the actual items that are being secured, the type of item (such as menustring), and a propagation flag that sets permissions for sub-elements of the item

you're securing. "Format of the Security Application File" on page 45 shows an example of a Security Registration File (SRF).

You must register any resources you consider sensitive, such as MenuBar menustrings, context menu menustrings, Tool Window items, class methods, and command line user executables.

3. Put the SRF in the directory /usr/OV/security/$LANG/Domains/registration on the NetView for AIX **server** machine (do not put the SRF on clients). Set $LANG=C for English language. For consistency, it is suggested that you use the same name for the SRF and for the Application Registration File (ARF).

   If you use the c_arf2srf utility to create an SRF from the application's ARF, the SRF will be put in the correct directory automatically.

4. Determine if you need to code any security APIs. If your application is launched *only* from the NetView for AIX EUI menu bar and you do not need to use any other security services (such as auditing), you do not need to code API calls. For applications registered for the NetView for AIX EUI, the security server automatically checks the permissions of EUI menu items and greys out the item or takes it off the menu bar completely (depending on the permissions the user assigns it in the SRF).
5. If your application can be invoked from outside the EUI, or if it needs to use other security services (like auditing), code a call to nvs_isClientAuthorized() just after main() in your program. This API establishes a security context with the NetView for AIX security server, and determines whether the user attempting to start the application has any type of access to the application. This call returns a 'permit' or 'deny' response.
6. Optionally, code a call to nvs_getClientPerms() to allow your program to retrieve the actual bitmask permissions that the user has assigned for the resource. In other words, you can find out the exact permissions the user has been assigned for the resource, such as read-only or read-write-execute.
7. Code a call to nvs_SecErrMsg to return status messages from the other security API calls.
8. Optionally, code a call to nvs_Audit to write an entry to an audit logfile if a user changes configuration of your application. In the call, you define what values will be audited.
9. Code a call to nvs_deleteSecContext in your exit handler to release the security context.

Understanding the Security Registration File

NetView for AIX platform applications must register with the security server. The Security Registration File (SRF) identifies the functional entities that need to be secured. SRFs are complemented by NetView for AIX security APIs. When used in conjunction with the registration files, the APIs provide the additional support required to make your application an integral part of the NetView for AIX security framework. In a client/server environment, SRFs reside on servers only.

Default Security Registration Files are stored in the directory **/usr/OV/security/$LANG/Domains/registration**. To see examples of other SRF files that are shipped with NetView for AIX, look in this directory. The *$LANG* environment variable provides support for native language localizability. If not otherwise defined, *$LANG* is assumed to be C, which means no native-language support is provided.

SRFs are created for different security groups after the network administrator defines security policies for the groups. NetView for AIX provides two pre-configured security groups: **Oper** and **SrAdmin**. These two security groups have their own SRF files in their security directories.

Format of the Security Application File

Here is an example of an SRF. Each field is explained in the list following this example.

```
###########################################################
#                                                         #
#    File for Network Management Security System         #
#                                                         #
# SRF seed file for xnmcollect                            #
#                                                         #
# Copyright International Business Machines Corp. 1995    #
# All rights reserved.                                    #
#                                                         #
###########################################################

DOMAIN_ID = xnmcollect
DESCRIPTION = " Defines and controls SNMP Data Collectors and Threshold Monitors"
SEPARATORS = ->
VALID_PERMISSIONS = rx
ELEMENTS =
    "xnmcollect" . FALSE  executable
    "SNMP Data Collector->Tools" . FALSE  menustring
    "SNMP Data Collector->Tools->Data Collection & Thresholds" . FALSE  menustring
    "SNMP Data Collector->Tools->Data Collection & Thresholds->SNMP..." . FALSE  menustring
```

The fields in the SRF are defined as follows:

DOMAIN_ID	Specifies the name of the application being registered with the security server. This is not the name of the specific menu item that you want to control; that is specified on the ELEMENTS block.
DESCRIPTION	Provides a one-line description of what the application does.
SEPARATORS	Identifies the character that is used as a separator when parsing the element names specified on the ELEMENTS block.
VALID_PERMISSIONS	Specifies the valid permissions that can be set for your element. In this example, valid permissions for the xnmcollect executable are *rx* (read and execute). Any single character from a to z inclusive is valid.
ELEMENTS	The actual elements being secured, and the security assigned to the elements and their sub-elements. In this example, the elements are the menu items for SNMP data collection, which is the xnmcollect executable.

Each element definition has the form **ELEMENTS=***element_name default_perm propagation type*. These fields are defined as follows:

element_name is the text of the item being secured. It can be in one of two formats:

- An application name as it appears on the menu bar, in the form `application name->menubar item->menustring`. In the sample SRF file, "SNMP Data Collector->Tools->Data Collection & Thresholds" is an example of this format. You must specify the application name and menubar item name for menu bar items.

  Note: The application name should be the same as the name under which the application is registered in the Application Registration File.

- An executable name. One or more names are separated by a separator charater and enclosed in double quotes. In the sample SRF file, "xnmcollect" is an example of this format.

  Note: If you used the nvs_isClientAuthorized or nvs_getCLientPerms APIs for your application, the executable name specified in the SRF must match exactly the target ID specified on these API calls.

default_perm is the access permissions set for the element. In a "seed" SRF file, this field should be set to a period (.), indicating that no permissions are assigned. The network administrator will set the permissions through the security dialogs after NetView for AIX is installed.

propagation specifies whether the permissions set for the element will be propagated to sub-elements under that element that are not specified in the SRF (such as Help). In this example, all propagations are set to FALSE.

type identifies the type of resource being controlled. Valid values are

- menustring - any string of text that appears on a menu pulldown or context menu
- executable - command-line executables
- method - any sensitive class method
- toolitem - any item that appears on the Tools Window

Creating Security Registration Files from Application Registration Files

Because much of the information in the SRF is already in an application's ARF, and because certain fields should match, it is a good idea to use the c_arf2srf utility to create the SRF for your application. This utility reads an application's ARF, creates an SRF using the application information, and places the SRF in the /usr/OV/security/$LANG/Domains/registration file.

Chapter 5. Designing Application Help

This chapter describes the NetView for AIX help system and how to use it. All applications that you create should provide online help information as described in this chapter. Users benefit from having a help system that behaves consistently across applications, even though those applications have been developed by more than one vendor or application developer.

There are three ways to provide help for users:

- Through the NetView for AIX help system

  Use the NetView for AIX help system to provide help from the NetView for AIX main menu bar and from dialog boxes. See “Using the NetView for AIX Help System” for more information.

- Through application-specific help

  You can construct custom help menus to provide users with detailed help for a specific application. See “Providing Help from Your Application's Menu” on page 54 for more information.

- Through the OVwShowHelp routine

  You can process help requests in your application and use this routine to present a help file or index. See “Displaying Help Information Programmatically” on page 56 for more information.

An important source of guidance on presenting help information is the *NetView for AIX Application Interface Style Guide*. This guide describes the types of help you can provide, as well as the recommended format for help entries. The *NetView for AIX Application Interface Style Guide* provides examples of each of the types of help described in this chapter.

Note: The NetView for AIX program has implemented a hypertext help system. This system is currently not available to application developers. All the capabilities described in this chapter are still fully functional in Version 4 of NetView for AIX. If you write applications, or if you build applications using the MIB Application Builder, use the help system described in this chapter.

Using the NetView for AIX Help System

The NetView for AIX help system can manage many of the tasks required to present help information to the user. In its most basic form, it is composed of the following items:

- A set of ASCII text files that contain the application help information.
- A mechanism for linking user actions, such as selection of a Help button or pull-down menu, to a particular help file.
- A viewing facility that presents help information to the user. The viewing facility, called ovhelp, is an X-based help system that displays help text in a window, using scroll bars to allow users to scroll through the text.

There are three different ways in which you can provide help to users of your application:

- You can provide help through the Task Index and Function Index, which are accessible from the `Help` pull-down menu on the NetView for AIX main menu bar.
- You can provide help through panels accessible from your application's dialog boxes.
- You can provide a `Help` pull-down menu on your application's menu bar, and show a list of topics on which help is available.

The following pages describe how to provide help in each of these ways.

Providing Help from the NetView for AIX Menu

Read this section if you want your application's help information to be available under the NetView for AIX program's main menu bar. The Help pull-down menu on the main menu bar contains the following menu items:

```
Indexes
NetView for AIX Help
NetView for AIX Library
On Help
Legend
AIX Base OS InfoExplorer*
On Version
```

You can add application help information directly under the **Tasks** and **Functions** Indexes. Other menu items, such as **On Version**, use information from application registration files. The remaining help menu entries are used only by the NetView for AIX program.

Table 5 shows the purpose of each item and indicates whether you can add help information under the item. Items that have "Reg" in the second column can be modified by changing a registration file. Items that have "Annot" in the second column can be modified by the user or administrator through the DynaText annotation function. For a fuller description of the items on the NetView for AIX main Help menu, refer to the *NetView for AIX User's Guide for Beginners*.

Table 5. Help Menu Item Descriptions

Help Menu Item	Can Developer Modify?	Description
`Indexes..Applications`	Reg	Provides version, copyright, and description information from each application's application registration file.
`Indexes..Tasks`	Yes	Provides information related to tasks performed with the application.
`Indexes..Functions`	Yes	Provides information on functions accessible from the menu bar or explodable symbols.
`NetView for AIX Help`	Annot	Provides task-oriented help on using the NetView for AIX program.
`NetView for AIX Library`	Annot	Invokes the DynaText program to present the NetView for AIX online manuals and any other libraries that have been created in DynaText format. This is a separately installable feature of the NetView for AIX program.
`On Help`	No	Describes each item in the Help menu.
`Legend`	Reg	Describes each symbol used by NetView for AIX and by applications. New symbols defined in symbol type registration files will be shown here.
`On Version`	No	Provides version information for each application from its application registration file.
`AIX Base InfoExplorer`	No	Invokes the InfoExplorer program to present the online manuals that describe the AIX operating system.

Through this menu, your user can view help information for any NetView for AIX applications that have created help files to appear in the Task and Function Indexes.

To integrate application information into the Task Index or the Function Index, place the ASCII text files in the appropriate subdirectories as described below. The NetView for AIX program automatically integrates the help information into the Help menu.

Providing Applications Index Help

The Applications Index lists all applications that are integrated with the NetView for AIX program. When the user clicks on an application name, the help system displays version, copyright, and descriptive information about the application. For examples of the items that appear in such an index, select **Help..Indexes..Applications** from the NetView for AIX main menu bar.

The information displayed through the Applications Index is derived from the application's application registration file (ARF). See “Specifying the Application's Version, Copyright, and Description” on page 28 for details about coding this information in your application's ARF.

Providing Tasks Index Help

The Tasks Index is an index of help topic titles associated with tasks the user performs through NetView for AIX applications. When the user clicks on a topic title, the topic information is displayed, complete with detailed documentation of the task. The following example, which was taken from the NetView for AIX Task Index, shows the types of items that might appear in the Tasks Index:

```
AddConnection
ConfigureSubmap
DescribeMap
FindHelp
ViewStatus
```

To add an entry to the NetView for AIX Tasks Index, follow these steps:

1. Construct an ASCII file containing task help information. The help file should follow a specific format outlined in the *NetView for AIX Application Interface Style Guide*.
2. Store the ASCII help file in the `/usr/OV/help/$LANG/<apphelpdir>/OVW/Tasks` directory.

As an example, assume that the help file is named `task_open.help`. Also, assume that the default value for *$LANG* is **C**, and the application registration file contains a HelpDirectory entry with the value map_app. The task_open.help file will be stored in the directory `/usr/OV/help/C/map_app/OVW/Tasks`.

If the help file is not stored in the correct directory, or if the first line of the help file is not formatted correctly, the ovhelp utility will not display the help topic in the Tasks Index.

The ovhelp utility constructs the Tasks Index from all files present in the Tasks directory. The Tasks Index contains a list of all task help files provided by applications. The ovhelp utility extracts the first line from each file in this directory and uses it as an entry in the Tasks Index. Help files will be sorted according to the first line, so choose the titles carefully to place your files where you want them in the index. The remainder of the file is treated as the task help, and is presented to the user when the task is selected from the index.

You can place task entries either in the NetView for AIX Tasks Index on the main menu bar or in a Tasks Index on an application-specific menu bar. Use the NetView for AIX Tasks Index for the most general tasks that might be performed by the application's users. Leave detailed descriptions of tasks for the application's private Tasks Index. Refer to the *NetView for AIX Application Interface Style Guide* for more information.

Providing Function Index Help

The Functions Index is an index of help topics associated with the functions of NetView for AIX applications. The Functions Index contains a list of all function help files provided by applications. These files should describe the application's operating capabilities that are accessible through executable symbols or menu items. If your application associates actions with executable symbols, or adds to or modifies the NetView for AIX menu structure, add entries to the NetView for AIX help menu Functions Index. Describe the additional or modified entries as well as any extensions you have made to the operation of the NetView for AIX program.

Clicking on a topic in the index displays information on the function of an executable symbol or menu item. Each function component should be documented in terms of what it does and how it works. The following example shows the types of items that might appear in the Functions Index:

```
File→DeleteMap...
File→DescribeMap
File→NewMap...
File→Exit
```

Each entry contains the name of the main menu entry, followed by an arrow and the function name.

To add an entry to the NetView for AIX Functions Index, follow these steps:

Step 1. Construct an ASCII file containing function help information. The help file should follow a specific format outlined in the *NetView for AIX Application Interface Style Guide*.

Step 2. Store the ASCII help file in the directory, /usr/OV/help/$LANG/<apphelpdir>/OVW/Functions.

As an example, assume that the help file is named `fun_open.help`. Also, assume that the default value for `$LANG` is `C`, and the application registration file contains a HelpDirectory entry with the value `map_app`. The `fun_open.help` file will be stored in the directory, /usr/OV/help/C/map_app/OVW/Functions.

If the help file is not stored in the correct directory, or if the first line of the help file is not formatted correctly, the ovhelp utility will not display the help topic in the Tasks Index.

The ovhelp utility constructs the Functions Index from all files present in the Functions directories. The Functions Index contains entries for all applications that provide function help files. The ovhelp utility extracts the first line from each file in this directory, and uses it as an entry in the Functions Index. The remainder of the file is treated as the function help and is presented to the user when the function is selected from the Functions Index. Help files will be sorted according to the first line, so choose the titles carefully to place your files where you want them in the index.

Use the NetView for AIX Help Functions Index to describe the functions your application adds to the NetView for AIX menu structure. Do not describe application-specific menus in the Help Functions Index. Refer to the *NetView for AIX Application Interface Style Guide* for guidance in creating your own application-specific menu structure.

Providing Help through a Dialog Box

Another way users can get help is through Help buttons in dialog boxes managed by the NetView for AIX program. When you create the application registration file for your application, code a block called an Enroll block. Enroll blocks are used by the NetView for AIX program to construct dialog boxes to interact with the user when the following functions are selected:

- Adding an object to the map
- Connecting two symbols on the map

- Changing an application's configuration
- Changing an object description

If your application uses Enroll blocks to create dialog boxes, you can use the HelpFile entry in the Enroll block to specify the help text file for your dialog box. If the user presses the Help button on that dialog box, the NetView for AIX program displays the appropriate help file using the ovhelp file viewer.

To add a help entry to a dialog box, follow these steps:

Step 1. Construct an ASCII file containing dialog box help information. The help file should follow a specific format outlined in the *NetView for AIX Application Interface Style Guide*.

Step 2. Store the ASCII help file in the `/usr/OV/help/$LANG/<apphelpdir>/OVW/Dialogs` directory.

Step 3. Link the help file to the particular dialog box that it represents. Linking is done by specifying the path of the help file in the Enroll block in the application registration file using the HelpFile entry. The file path is specified relative to the `<apphelpdir>` directory.

The syntax of the HelpFile entry is described in Chapter 3, "Creating and Using the Application Registration File" on page 27.

The following example shows how to link the help file to the Enroll block in the application registration file. Assume that the help file is named `dlg_add_IP.help`.

```
Application "My app" {
   ...
   HelpDirectory thisapp;
   ...
   Enroll Describe {
      if isNode {
         HelpFile "OVW/Dialogs/dlg_add_IP.help";
         ...
      }
   }
}
```

Providing Help from Your Application's Menu

The help integration methods described earlier are not adequate for all applications. If your application provides a customized user interface, such as an application menu, you will need to use more advanced help integration methods to provide additional help for your users.

This section explains how to provide help information for applications that provide a customized user interface. Help information that is not provided through the Help pull-down menu on the NetView for AIX main menu bar, or through NetView for AIX-managed dialog boxes, is considered application-specific and must be presented by the application.

The NetView for AIX program provides limited support for applications that communicate with the user directly through a customized interface. It is your responsibility to construct the required help menus and dialog boxes using X programming primitives. You must also determine when a user needs help information. Once a user's request for help has been detected, the application can call an NetView for AIX API library routine to display the help information.

Including Help in a User Interface

The *NetView for AIX Application Interface Style Guide* refers to applications that provide their own interfaces as application tools and application subtools. *Application tools* are a set of integrated functions within an application that is presented to the user in one or more windows (for example, software that manages user accounts). *Application subtools* are specific functions that can be carried out from an application tool (for example, adding or deleting user accounts). If you want application tools and subtools to provide help access, you must implement it.

If your application provides an application-specific user interface (for example, an application tool window), provide access to application help information through that interface. Before designing your user interface, refer to the *NetView for AIX Application Interface Style Guide* for information on how to structure menus and dialog boxes for application tools and subtools. Refer to the *NetView for AIX Application Interface Style Guide* for recommendations about providing help before designing your user interface.

Providing Application-Specific Help Files

You should provide application-specific help text files using the formats recommended in the *NetView for AIX Application Interface Style Guide*. You should store your application-specific help files under your help directory, using subdirectories to partition function, task, and dialog help files. You can use the following directory structure for your application-specific help files. Note that these directories are different from those used for the previously described NetView for AIX help files.

```
/usr/OV/help/$LANG/<apphelpdir>/Functions
/usr/OV/help/$LANG/<apphelpdir>/Tasks
```

Developing the Help Directory Structure

Any time a user requests help, the NetView for AIX program looks in a predefined directory structure for the appropriate help information. The help information is stored in ASCII text files in subdirectories under the following directory path:

```
/usr/OV/help/$LANG/<apphelpdir>
```

The *$LANG* environment variable defines the default language for help files. The default value of *$LANG* is **C**.

Recall that applications can specify a help directory in the application block in the application registration file. That value is substituted for `<apphelpdir>`. For example, if your default language variable is `C` and your application registration file contains the entry:

```
Application "My app" {
   ...
   HelpDirectory "thisapp";
   ...
}
```

the root of your help directory will be:

```
/usr/OV/help/C/thisapp
```

All help files are stored relative to this directory path. You should substitute your own values for `$LANG` and `<apphelpdir>`.

Refer to the *NetView for AIX Application Interface Style Guide* for guidance in formatting the help files that apply to your application. You can also review the help files provided with other NetView for AIX applications to see examples of the help file format.

Displaying Help Information Programmatically

Your application can use the OVwShowHelp() routine to display either:

- The contents of a help file through the ovhelp file viewing facility.
- A help index from which users can select a particular help topic.

The OVwShowHelp() routine has the following format:

```
int OVwShowHelp( unsigned long helpType, char *helpRequest );
```

It has the following arguments:

- The `helpType` argument indicates whether a help file or a help index should be displayed. It can have the values ovwHelpFile or ovwHelpIndex.
- The value of `helpRequest` can vary, depending on the help type. If *helpType* is ovwHelpFile, the `helpRequest` argument contains the path of a specially-formatted help file. The path is specified relative to the root of the application's help directory. Recall that the application defines a help directory with a HelpDirectory entry in the application registration file.

If *helpType* is `ovwHelpIndex`, the `helpRequest` argument contains the path of a help index file. The ovhelp man page describes how to construct and build index files.

Whether you provide help through the NetView for AIX main menu bar, through NetView for AIX-managed dialog boxes, or through an application-specific user interface, providing help information with your application will improve its usability and your users' productivity and satisfaction.

Part 2. Working with the NetView for AIX User Interface

Chapter 6. Understanding the NetView for AIX User Interface

This chapter describes design concepts and coding techniques for writing applications that interact with the NetView for AIX graphical user interface. After reading this chapter, you will know how to initialize your application, how to structure your application, and how to perform several basic interface-related tasks within your application.

Many aspects of your application's behavior are specified in the Application Registration File that you create for your application. The other building block for your application is the library of routines called the NetView for AIX APIs. Use these routines to take advantage of the programming interfaces offered by the NetView for AIX program.

In writing applications to interact with the NetView for AIX graphical user interface, you will use the routines of the NetView for AIX EUI (End-User Interface) API. This API contains over 200 routines, organized into groups according to their functions. Though you can use the full breadth of the NetView for AIX EUI API, a basic understanding of a few key routines in each group will allow you to implement sophisticated applications.

There are seven groups of EUI API routines, organized as follows:

Application integration	There are a small number of routines that integrate your application with the NetView for AIX graphical interface. Every application that uses the EUI API needs to use at least some portion of these calls. These routines allow your application to connect to the NetView for AIX program, to determine when users select particular menu items, and to handle errors. Several of these routines are described in this chapter.
Object database access	There are over 40 EUI API routines that operate on objects and fields. These routines are used to create objects, to create the fields that comprise objects, to get or set field values in objects, and to relate fields to objects and objects to fields. See Chapter 7, "Creating and Using Objects and Fields" on page 79 for more information on these routines.
Symbol routines	These routines create symbols, alter symbol behavior and appearance, and get information about an object as it exists on a map. See Chapter 8, "Creating and Using Symbols" on page 99 for more information on these routines.
Map and submap routines	The EUI API contains many routines that are used to operate on maps and submaps. These routines permit you to create and modify submaps, as well as to retrieve information about maps and submaps. See Chapter 9, "Creating and Using Submaps" on page 131 for more information on these routines.

User verification	There are 5 EUI API routines that allow a program to verify changes that a user attempts to make to maps and objects through the graphical user interface. These routines are described in “Participating in Map Changes” on page 152.
Dynamic registration	There are over 60 EUI API routines that dynamically configure the NetView for AIX menu structure. A list of these routines appears later in this chapter.
Callback routines	The NetView for AIX program uses callback routines to communicate with applications when various events occur. The NetView for AIX EUI API provides many definitions for callback routines to be provided by the developer. Callback routines are described in this chapter.

If your application uses any of the routines in the NetView for AIX EUI API, except the object database routines, you must create an application registration file for your application. See Chapter 3, “Creating and Using the Application Registration File” on page 27 for information about application registration files.

Structuring Your Application

The event-driven model of programming was described in “Developing Applications for NetView for AIX” on page 7. To use this model, code your application to perform the following three steps:

Step 1. Connect to the NetView for AIX program.
Step 2. Define callback routines to be invoked when specific events occur.
Step 3. Enter a loop that waits for and processes events.

Connecting Your Application to the NetView for AIX Program

The first step is to connect your application to the NetView for AIX program. To do this, you must call the OVwInit() routine. The OVwInit() routine initializes internal API data structures and establishes a connection from your application to the NetView for AIX graphical user interface. No other EUI API calls can occur before your application connects to the NetView for AIX interface. After the OVwInit() call has been issued, you are free to issue any other calls in the EUI API.

In designing your application, determine what data your application must have before it proceeds. For example, it may need the name of the open map or the open submap. Note that when your application is invoked, a map has already been opened, so you cannot count on the ovwMapOpen event to identify the open map. “Getting Map and Submap Information” on page 138 describes how your application can retrieve information about the open map and other information it may require.

When your application has finished interacting with the NetView for AIX program, you must disconnect from the NetView for AIX graphical user interface. This is done by calling the OVwDone() routine. The OVwDone() routine cleans up internal API data structures and closes the connection from the application to the NetView for AIX graphical user interface. An example code segment that illustrates the use of the OVwInit() and OVwDone() routines appears in the callback routine example on page 65.

Defining Callback Routines for Events

The second step that every NetView for AIX application should perform is to define callback routines that are invoked when specific events occur. Events can be caused by user actions, such as adding a symbol to a map, or by applications, such as creating a submap or changing symbol status.

Applications do not receive all NetView for AIX events automatically. Applications must specifically register callback routines for events in which they are interested. If an event occurs and a callback routine is not registered for it, the event is not sent to the application. Some symbol-related events are sent only to applications that have registered interest in the affected symbol.

Waiting for and Processing Events

The third step that an NetView for AIX application should perform is to enter a loop that waits for notification of an event. When an event notification is received, the callback routine for that event is invoked. The technique you use for recognizing and processing events depends on your application design and on the types of events your application must process. See “Processing Events” on page 67 for a description of these techniques.

NetView for AIX Events

The NetView for AIX program defines 36 events for application use. These events are listed in Table 6.

Table 6 (Page 1 of 2). NetView for AIX Events

Event	Description
ovwEndSession	NetView for AIX EUI session termination
ovwSelectListChange	Map selection list changed
ovwMapOpen	Map open
ovwMapClose	Map close
ovwQueryAppConfigChange	Application configuration change query
ovwConfirmAppConfigChange	Application configuration changed
ovwQueryDescribeChange	Description change query
ovwConfirmDescribeChange	Description changed
ovwQueryAddSymbol	Query to add symbol to map
ovwConfirmAddSymbol	Symbol added to map
ovwQueryConnectSymbols	Query to connect symbols
ovwConfirmConnectSymbols	Symbols connected
ovwQueryDeleteSymbols	Query to delete symbols
ovwQueryDeleteSubmap	Query to delete a submap
ovwConfirmDeleteSymbols	Symbols deleted from map
ovwConfirmDeleteObjects	Objects deleted from map
ovwConfirmDeleteSubmaps	Submaps deleted from map
ovwConfirmCreateSymbols	Symbols created on map

Table 6 (Page 2 of 2). NetView for AIX Events

Event	**Description**
ovwConfirmCreateObjects	Objects created on map
ovwConfirmCreateSubmaps	Submaps created on map
ovwConfirmMoveSymbol	Symbol moved
ovwConfirmManageObjects	Objects managed
ovwConfirmUnmanageObjects	Objects unmanaged
ovwConfirmHideSymbols	Symbols hidden
ovwConfirmUnhideSymbols	Symbols unhidden
ovwConfirmSymbolStatusChange	Symbol status change
ovwConfirmObjectStatusChange	Object status change
ovwConfirmCompoundStatusChange	Compound object status change
ovwConfirmCapabilityChange	Object capability field change
ovwConfirmAcknowledgeObjects	Object acknowledged
ovwConfirmUnacknowledgeObjects	Object unacknowledged
ovwConfirmCreateMetaConnection	Metaconnection created (xxmap only)
ovwConfirmExplodeObject	Object exploded
ovwUserSubmapCreate	Submap Creation Notification
ovwSubmapOpen	Submap open
ovwSubmapClose	Submap closed

Note: The OVwSubmapClose event is issued only when a submap is closed from the navigation tree. It is not issued when the window containing a submap is closed. If your application will depend on this event, be sure to tell your users to close submaps from the navigation tree.

Registering for Events

You can register your application for as many events as it requires. To register for events, use the OVwAddCallback() routine, which has the following function prototype:

```
int OVwAddCallback( OVwEventType event,
                    OVwFieldBindList *capabilitySet,
                    OVwCallbackProc callbackProc,
                    void *userData );
```

The following arguments must be supplied to the OVwAddCallback() routine:

event	Defines the event to be registered.
capabilitySet	Used for callback routines that pertain to objects. This field lets you associate callback routines not only to a specific event, but also to specific kinds of objects based on the values of capability fields. The callback routine will be called only for objects that have the specified capability field values. This filtering can reduce the number of events an application receives. You can have several callback routines within your application registered for the same event, provided that they use different object capability fields.

	Code NULL if object capability fields are not of interest to your callback routine.
callbackProc	The name of the application's callback routine.
userData	A user-defined, user-supplied parameter for the callback procedure. You are free to use this field however you wish.

The OVwAddCallback() routine is used to register all events; however, you can change the arguments depending on the event type. The NetView for AIX program passes different types of information to the callback routine based on the type of event that occurs. In some cases, the NetView for AIX program might need to pass object information, while in another, it might need to pass symbol information. A specific callback procedure function prototype exists for each event type. These function prototypes are found toward the end of the <OV/ovw.h> header file.

The following example shows how an application connects to the NetView for AIX program and how it registers a callback routine. In this example, the application registers for notification when the NetView for AIX EUI session is terminated.

```
#include <OV/ovw.h>

endSessionCB( userData, type, normalEnd )
void *userData;
OVwEventType type;
OVwBoolean normalEnd;
{
   printf( "NetView for AIX is terminating, so are we\n" );
   OVwDone();
   exit(0);
}

main()
{
   int ret;

   if ( OVwInit() < 0 ) {
      printf( "application couldn't initialize with
      NetView for AIX\n" );
      exit( 1 );
   }

   ret = OVwAddCallback( ovwEndSession, NULL,
                        (OVwCallbackProc) endSessionCB, NULL );
   if ( ret < 0 ) {
      /* error processing */
   }
   OVwMainLoop();
}
```

Note: The OVwMainLoop() procedure in the main() function has not yet been described. It will be explained in “Checking for NetView for AIX Events” on page 68.

Action Events

In addition to receiving events, NetView for AIX applications can also receive notification when users invoke application-provided actions, either from NetView for AIX menu items or from executable symbols. These are special events called action events. Callback registration for action events is very similar to event registration described previously. Action events are registered with the OVwAddActionCallback() routine, which has the following function prototype:

```
int OVwAddActionCallback( char *actionID,
                     OVwActionCallbackProc callbackProc,
                     void *userData );
```

Specify the following arguments:

actionID	An action name from the application registration file with which a callback routine will be associated.
callbackProc	The name of the callback routine.
userData	A pointer to a user-defined, user-supplied parameter for the callback procedure. Use this field as you need it.

All action event callback routines have the following function prototype:

```
void (*OVwActionCallbackProc)( void *userData,
           char *actionID, char *menuitemID, OVwObjectIdList *selections,
           int argc, char **argv, OVwMapInfo *map,
           OVwSubmapId submapID );
```

Specify the following arguments for an action event callback routine:

userData	A user-defined parameter that may have been specified on the OVwAddActionCallback() procedure call.
actionID	The string by which the action is known in the application registration file.
menuitemID	A string containing the label of the menu item used to invoke the application. NULL indicates that the application was invoked by an executable symbol.
selections	A pointer to a copy of the current NetView for AIX selection list. Selection lists are described in “Using the Object Selection List” on page 69.
argc, argv	Contain callback arguments as defined in the CallbackArgs statement in the ARF.
map	A pointer to an OVwMapInfo data structure that contains information about the open map. The OVwMapInfo structure is described in “Getting Map Information” on page 138.
submapID	The submap ID of the submap on which the event occurred.

The following example shows how to register for Action events. Assume this entry is present in the application's application registration file:

```
Application "My application" {
     MenuBar "Configure" {
         "Callback Test" f.action "OVwAddActionCallback test";
     }

     Action "OVwAddActionCallback test" {
         Command "<your executable path>";
     }
 }
```

The following example shows how to register a callback routine to handle user selection of the menu item:

```
#include <OV/ovw.h>

myActionCB( userData, actionID, menuitemID,
            selections, argc, argv, map, submap )
void *userData;
char *actionID;
char *menuitemID;
OVwObjectIdList *selections;
int argc;
char **argv;
OVwMapInfo *map;
OVwSubmapId submap;
{
 ...
}

main() {
   int ret;

   if ( OVwInit() < 0 ) {
      printf( "error initializing with OVw\n" );
      exit( 1 );
   }

   ret = OVwAddActionCallback("OVwAddActionCallback test",
                             (OVwActionCallbackProc) myActionCB, NULL );
   OVwMainLoop();
}
```

Processing Events

There are three basic techniques for processing events in NetView for AIX applications. You can use any of the following methods:

- Process only NetView for AIX events (using OVwMainLoop()).
- Process both X and NetView for AIX events (using OVwXtMainLoop()).
- Use your own select() loop to process events.

Your application needs will dictate which technique to use. If your application does not use Xt calls directly, the first technique will probably be the easiest to use. If your application uses Xt calls, you will probably need to use the second technique (using OVwXtMainLoop()). If neither of these two general-purpose routines suffice, you can use the third method and implement your own event-processing loop. The remainder of this section describes these techniques in more detail.

Checking for NetView for AIX Events

The simplest way to check for NetView for AIX events is to use the OVwMainLoop() routine.

OVwMainLoop() loops forever, continually processing NetView for AIX registered events. This routine checks each NetView for AIX event against a list of preregistered events, and, if a callback routine is registered for the event, the callback routine is invoked. Previous examples in this chapter have shown how to use the OVwMainLoop() routine. You may have noticed in the examples that the last statement in each main() routine is a call to OVwMainLoop(). Because OVwMainLoop() executes indefinitely, the main() routine does not exit. NetView for AIX applications must exit through another routine.

Processing X Events and NetView for AIX Events

NetView for AIX applications that perform X-event processing should not use the OVwMainLoop() routine. Rather, they should use the OVwXtMainLoop() routine to process events. The OVwXtMainLoop() EUI API routine is designed specifically to include both NetView for AIX event processing and X-event processing within a single event handling call. Sample code that demonstrates X-event processing can be found in the file, /usr/OV/prg_samples/ovw_examples/app6/six.c.

Checking for File Descriptor Input

By default, the OVwMainLoop() and OVwXtMainLoop() routines process only NetView for AIX and X events. As a convenience, the NetView for AIX EUI API lets you extend these routines to also check for and process file descriptor input events. This enables your application to listen for messages from other applications. The OVwAddInput() routine adds an application file descriptor to the NetView for AIX event-processing mechanism as another source of events. The routine has the following syntax:

```
OVwInputId OVwAddInput( int file_descriptor, int conditionMask,
                        OVwInputCallbackProc proc, void *userData );
```

OVwAddInput() is passed the source file descriptor, a condition mask, and an application-specific user data parameter to be sent to the callback routine. The callback routine has the following function prototype:

```
void (*OVwInputCallbackProc)( int fileDescriptor, void *userData );
```

The following example shows how to use the OVwAddInput() routine:

```
#include <OV/ovw.h>

my_fd_event_CB( fileDescriptor, userData )
int fileDescriptor;
void *userData;
{
 ...
}

main()
{
   int fd;
   OVwInputId ret;

   if ( OVwInit() < 0 ) {
      printf( "error initializing with NetView for AIX\n" );
   }
   /* <open socket 'fd', connect to a remote
      system and wait for input> */
   ret = OVwAddInput( fd, ovwReadMask, my_fd_event_CB, NULL );
   OVwMainLoop();
}
```

Using a Select() Loop

If your application has special event processing needs that are not met by OVwMainLoop() or OVwXtMainLoop(), you can process events using select() directly. You can use the OVwFileDescriptor() routine to obtain the file descriptor associated with NetView for AIX event processing. Once you have the file descriptor, you can add it to the select mask in your select() processing loop.

Checking the Event Queue Manually

There may be occasions where your application is performing long, intense operations under the assumption that NetView for AIX map status has not changed in the interim. Critical events can occur that might justify the interruption of your processing. One such event is the ovwMapClose event.

You can manually check the NetView for AIX event queue for the presence of specific events using the OVwPeekOVwEvent() routine. This routine is called with a specific event as an argument. The routine nondestructively examines the NetView for AIX event queue and returns a boolean value indicating the presence of the event.

Using Advanced Event Queue Management

A number of other advanced mechanisms are also available to check for NetView for AIX events. The OVwPending() and OVwProcessEvent() routines provide low-level control over NetView for AIX event processing and are not described here. If the basic NetView for AIX event checking calls are not adequate for your application, you can refer to the man pages for information on these advanced calls.

Using the Object Selection List

The object selection list is the primary means for users to pass arguments to NetView for AIX applications. The selection list is automatically provided as an input argument to action callback routines when they are invoked. This is the standard mechanism used by applications to receive the contents of the object selection list. Some applications, however, need to determine the contents of the object selection list at other times. The EUI API provides the OVwGetSelections() routine for this purpose. It has the function prototype:

```
OVwObjectIdList *OVwGetSelections( OVwMapInfo *map, char *actionId );
```

The OVwGetSelections() routine returns a pointer to an OVwObjectIdList structure that contains:

- A count of the number of selected objects in the list. The selected objects must satisfy the selection rules for the action specified as actionId.
- A pointer to a contiguous area of memory containing object IDs. The memory may be treated as an array.

Most EUI API list data structures are implemented using a pointer to a contiguous area of memory, rather than a NULL-terminated linked list.

The following example shows how to use the OVwGetSelections() routine to traverse the list of objects whose IDs are represented in the selection list. This example uses the OVwGetMapInfo() routine to get the map information. See “Getting Map Information” on page 138 for more information.

```
#include <OV/ovw.h>
 ...
main()
{
   int i;
   OVwObjectIdList *op;
   OVwMapInfo *map;
   OVwObjectId *lp;
     ...
   map = OVwGetMapInfo();
   op = OVwGetSelections( map, NULL );
   printf( "There are %d objects in the selection list\n", op->count );
   for ( i=0, lp = op->object_ids; i<op->count; i++, lp++ )
      {
      printf( "object id[%d] is %ld\n", i, *lp );
      }
   OVwDbFreeObjectIdList( op );
      ...
}
```

Note: If an application has been invoked from an object menu, the selection list passed into the application will consist of the object from which the object menu was selected. Using OVwGetSelections() in this case will return the list of objects selected by the user before invoking the application, rather than the object from which the object menu was selected.

Selection Rules and the Selection List

When you create the ARF for your application, you can define selection rules for entries in the selection list. Selection rules are defined within the context of a particular action and determine whether the action will be available based on the list of selected objects. Selection rules in the ARF are available for programmatic use as well.

To use selection rules in NetView for AIX programs, use the action ID associated with the selection rule in the ARF as an input argument to the OVwGetSelections() routine. For example, consider the following ARF:

```
Application "My app"
{
 ...
   Action Trends {
   SelectionRule isNode && isDevice;
 ...
}
 ...
}
```

To use the above selection rule in your program, you pass the *Trends* argument to your OVwGetSelections() call. The OVwGetSelections() routine returns the current selection list only if the selection rule in the Trends action is valid for all the objects in the selection list.

The ovwSelectionListChange Event

The NetView for AIX program generates the ovwSelectionListChange event whenever the selection list changes. You can register a callback routine that uses OVwGetSelections() to determine the current selection list when this event is received. Because selection list changes can occur frequently, register for the ovwSelectionListChange event only if it is absolutely required. Excessive use can degrade system performance.

Some Useful EUI Techniques

This section describes several techniques that you can use to accomplish common user interface-related programming tasks.

Managing Memory with the EUI API

Many NetView for AIX EUI API calls allocate data structures in memory. You should be careful to free the memory associated with these structures when your program no longer needs them. Convenience routines are provided for this purpose. For example, the OVwListSubmaps() routine creates an OVwSubmapList data structure, which you should free with a call to the OVwFreeSubmapList() routine. The man pages for API routines that create data structures list the name of the associated routine that frees that structure.

Some routines do not have accompanying memory-freeing routines. These routines return dynamically-allocated character strings. You must use the C-language **free()** function to free the memory allocated for these strings. Use **free()** to free the results of the following API routines:

- OVwCreateMenuItem
- OVwGetRegContext
- OVwGetMenuPathSeparator
- OVwGetMenuItemPath
- OVwGetMenuItemMenu
- OVwGetFirstRegContext
- OVwGetNextRegContext
- OVwGetNextMenuItem
- OVwGetFirstMenuItem
- OVwGetFirstAction
- OVwGetNextAction
- OVwFindMenuItem

- OVwDbObjectIdToSelectionName
- OVwDbObjectIdToHostname
- OVwDbGetUniqObjectName
- OVwDbGetFieldEnumByName
- OVwDbGetFieldStringValue
- OVwDbFieldIdToFieldName
- OVwDbGetEnumName
- OVwGetAppName

Retrieving a Routine's Error Code

NetView for AIX stores an internal error code for every NetView for AIX API call you make. This internal error code contains an integer value that represents either success or, if an error occurs, the cause of the error. You can request this value by calling the OVwError() EUI API routine, which returns the error code of the last OVw routine called by the application. All error values are defined in the file <OV/ovw_errs.h>. The man page for each EUI API routine describes how it returns errors.

Converting an Error Code to a String

The EUI API also provides a routine that converts an integer error code into the corresponding text string description. The OVwErrorMsg() routine returns a pointer to a character string in static memory that describes the error. The call has the following syntax:

```
char *OVwErrorMsg( int error );
```

Because the NetView for AIX program allocates the memory for the character string from a static buffer, you should not attempt to free the string memory after using it. The following example shows one way to use the OVwError() and OVwErrorMsg() routines:

```
#include <OV/ovw.h>
 ...
main( argc, argv )
   int argc;
   char **argv;
   {
   if ( OVwInit() < 0 ) {
      if ( OVwError() == OVw_OVW_NOT_RUNNING ) {
         printf( "NetView for AIX must be running prior to
         running this application\n" )
      }
      else {
         printf( "%s\n", OVwErrorMsg( OVwError() ) );
         }
      }
    ...
}
```

Checking NetView for AIX IDs

Upon completion, many EUI API routines return an ID. ID examples include object IDs, field IDs, symbol IDs, and submap IDs. As a convenience, the EUI API provides macros that test IDs in different ways. They have the form:

```
OVwBoolean OVwIsIdNull( id );
OVwBoolean OVwIsIdEqual( id1, id2 );
```

Though you could bypass these macros and implement these tests yourself, these macros hide the underlying ID data types, making your program more portable in the event that an ID implementation changes in the future.

Getting your Application Name

The Application block within the application registration file defines your application name. A number of EUI API calls require an application name as an input argument. If you hard-code those EUI API calls to use the application name defined in your application registration file, changing the application name in the ARF would cause your application to fail.

This problem is solved by using a routine in the EUI API that retrieves your application name from the ARF. The OVwGetAppName() routine returns the application name as defined in the ARF. The following code segment demonstrates:

```
char *myname;
 ...
myname = OVwGetAppName();
printf( "My name is: %s\n", myname );
free( myname );
 ...
```

Note that the OVwGetAppName() routine returns a pointer to a string whose memory is dynamically allocated. You should free the memory when it is no longer needed.

Highlighting Objects

Applications can highlight one or more objects on a map as a result of a user-initiated action (e.g., identifying all objects with a particular characteristic). All symbols representing highlighted objects are graphically displayed with symbol labels in reverse video. Users can select highlighted objects with the **View..Select Highlighted** menu item to make them input for another operation. The highlighting routines have the following function prototypes:

```
int OVwHighlightObject( OVwMapInfo *mapInfo, OVwObjectId object,
                        OVwBoolean clearPrevious );
int OVwHighlightObjects( OVwMapInfo *mapInfo, OVwObjectIdList *objectList,
                         OVwBoolean clearPrevious );
```

The object ID or list of object IDs is passed as an argument. Object ID lists are described in “Using the Object Selection List” on page 69. The *mapInfo* parameter may be NULL, in which case these routines will refer to the open map. The *clearPrevious* flag controls whether previously highlighted objects are cleared. If this flag is set to false, previously highlighted objects remain highlighted. If it is set to true, previously highlighted objects are cleared. An application should use the *clearPrevious* flag once at the beginning of each action for which the highlighting is being done. If the application is performing highlighting in successive calls, then it should use the *clearPrevious* flag only for the first call.

Using Dialog Boxes

The user can perform a wide variety of operations through the graphical user interface. In many cases, the NetView for AIX program can handle the user operation without requiring any help from the application. In other cases, the NetView for AIX program might need to request more specific information to be passed to the application. A dialog box is used for this purpose.

There are four operations that might require your application's involvement:

- Adding a symbol to a submap
- Connecting two symbols on a submap
- Modifying an object's attributes
- Changing how an application is configured to operate on a map

These four operations are treated specially by the NetView for AIX program. The NetView for AIX EUI API acts as a mediator between the user and the application, enabling the application to control whether the operations are allowed. If the operations are allowed, the NetView for AIX program makes the changes on behalf of the user and then informs the application.

The NetView for AIX program provides special assistance to developers who support these operations in their applications. The NetView for AIX program does not require that you implement X-Windows System based dialog boxes for these operations. Rather, the NetView for AIX program lets you define the structure of these dialog boxes using entries in the application's application registration file. Using entries called Enroll blocks, developers can define the structure and behavior of these NetView for AIX-generated dialog boxes. The use of Enroll blocks is described in “Defining Dialog Boxes with the Enroll Block” on page 38.

Coding an Enroll block defines the appearance and function of your dialog box, but it does not present your dialog box to the user when the action is invoked. To have your dialog box appear when the user edits the map, you must register a callback for the editing event. This enables you to accept user actions by participating in the query-verify-confirm sequence. This process is described in “Participating in Map Changes” on page 152.

Developing Applications for Control Desk Windows

The NetView for AIX graphical user interface provides a special type of window called a *control desk* window. These windows give you more direct control of your application's execution than regular NetView for AIX windows permit. Control desk windows are typically used for applications that will be kept active over a period of time or for applications of which multiple copies may be active at one time. For example, the window in which the events display appears is a control desk window.

If you want your application to run in a control desk window, use XnvApplicationShell widgets to create windows that will be placed into a control desk. You must code several widget resources to enable the application to interact properly with the graphical user interface. Within the XnvApplicationShell class, set the following values:

Resource	Required Value
XnvNeuiManaged	This resource must be TRUE for a control-desk application.
XnvNoutside	This resource specifies the initial application placement. FALSE places the application in a control-desk window; TRUE places it outside a control-desk window.
XnvNassociatedShell	If your application creates additional windows, this resource must be set inside each secondary shell to the value of the application's main shell.

You can set these values in your application's Xdefaults file. To do this, remove the `XnvN` from the beginning and use the remainder of the name. You can also set these values using XtSetArg or XtSetValues.

Using the EUI API Help Routines

Most applications can integrate help information into the NetView for AIX help system without resorting to programming. Some applications, however, must use the EUI API to integrate application-specific help information into the NetView for AIX program. This section describes how to use the EUI API to incorporate help information into the NetView for AIX program.

A single EUI API routine is used to programmatically access and display help information. The OVwShowHelp() routine can display to the user either a help file or an index of help topics. The routine has the function prototype:

```
int OVwShowHelp( unsigned long helpType, char *helpRequest );
```

It has the following arguments:

- The *helpType* argument indicates whether a help file or a help index should be displayed. It can have the values **ovwHelpFile** or **ovwHelpIndex**.
- The *helpRequest* argument's value can vary, depending on the help type. If *helpType* is **ovwHelpFile**, then the *helpRequest* argument contains the path of a specially-formatted help file. The path is specified relative to the root of the application's help directory. Recall that the application defines a help directory with a *HelpDirectory* entry in the application registration file.

If the *helpType* value is ovwHelpIndex, the *helpRequest* argument contains the path of a help index file. The ovhelp() man page describes how to construct and build index files. Refer to the *NetView for AIX Application Interface Style Guide* for guidelines about the format of help files.

Dynamic Menu Registration

Most developers can rely on the NetView for AIX registration files to configure where an application is placed in the NetView for AIX menu structure. Some developers, however, might need to programmatically alter the structure of the NetView for AIX menu bar, the object menu, or the Tools Window. Programmatic access to these menu structures, though not encouraged, is available to developers. This capability is called *dynamic menu registration*. If you need to alter the menu structure, refer to these man pages:

- OVwAddMenuItem()

- OVwAddMenuItemFunction()
- OVwAddObjMenuItem()
- OVwAddObjMenuItemFunction()
- OVwAddToolPalItem()
- OVwCreateMenu()
- OVwCreateMenuItem()
- OVwCreateObjectMenuItem()
- OVwLockRegUpdates()
- OVwSaveRegUpdates()

These man pages describe the functions involved in dynamic menu registration, and point to other man pages that provide further details. The following example illustrates the use of the dynamic menu registration routines:

```
#include <stdio.h>
#include <OV/ovw.h>
#include <OV/ovw_reg.h>

int main(int argc, char **argv)
{
   int                  rc;
   OVwBoolean           block;
   OVwMenuItemRegInfo   menuItem, menuItem1, menuItem2;
   char                 *menuItemID, *menuItem1ID, *menuItem2ID;
   char                 *menuID;

   block    = atoi[argv];

   menuItem.label = "Testing";
   menuItem.mnemonic = "t";
   menuItem.accelerator = NULL;
   menuItem.precedence = 1;

   menuItem1.label = "Ping";
   menuItem1.mnemonic = "P";
   menuItem1.accelerator = NULL;
   menuItem1.precedence = 2;


   if (OVwInit() < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      exit(1);
   }

   if ((rc = OVwLockRegUpdates(block)) < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   menuItemID = OVwCreateMenuItem(&menuItem);
   if (menuItemID == NULL)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   menuID = "test";
   if ((rc = OVwCreateMenu(menuID)) < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   menuItem1ID = OVwCreateMenuItem(&menuItem1);
   if (menuItem1ID == NULL)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   if (OVwAddMenuItem(menuID, &menuItem1ID) < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }


  OVwAddMenuItemFunction(menuItem1ID, ovwFnShell,
                  "aixterm -e /etc/ping ${OVwSelection1}");

   if (OVwAddMenuItem(NULL, &menuItemID) < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   OVwAddMenuItemFunction(menuItemID, ovwFnMenu, menuID);


   if (OVwSaveRegUpdates(FALSE) < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      OVwUnlockRegUpdates();
      exit(1);
   }

   if (OVwUnlockRegUpdates() < 0)
   {
      printf("%s\n", OVwErrorMsg(OVwError()));
      exit(1);
   }

   OVwDone();
   fflush(stdout);
   exit(0);
}
```

The following chapters explain how to use the NetView for AIX EUI API to work with fields, objects, symbols and submaps. Detailed information on all routines is provided in the *NetView for AIX Programmer's Reference* and the man pages.

Chapter 7. Creating and Using Objects and Fields

This chapter describes the NetView for AIX object database routines. With these routines, you can create and manipulate objects and fields in the NetView for AIX object database.

Topics in this chapter include:

- "Using the NetView for AIX Object Database"
- "Creating Fields"
- "Creating Objects"
- "Getting and Setting Object Field Values"
- "Deleting Objects"

Using the NetView for AIX Object Database

The NetView for AIX object database manages all object and field information for the NetView for AIX program. All NetView for AIX maps use this object database. You will use only the object database routines described in this chapter, which eliminates the need for you to code to a particular database implementation. This scheme also permits future substitution of a different underlying database without affecting applications.

The NetView for AIX object database is implemented as a stand-alone module that works in conjunction with the rest of the NetView for AIX program. Entries in the NetView for AIX object database persist across NetView for AIX sessions. Fields and objects created in one NetView for AIX session are available to all other applications in all NetView for AIX sessions.

Because the NetView for AIX object database is separate from the rest of the NetView for AIX program, you can access it without using the OVwInit() routine to connect to the NetView for AIX program. You can use a subordinate routine called OVwDbInit() to connect directly to the object database. The NetView for AIX object database routines all begin with the prefix `OVwDb`. The formats and data structure definitions for all field and object database routines are located in the file, `<OV/ovw_obj.h>`.

Creating Fields

Fields are object attributes stored in the NetView for AIX object database. Fields are the building blocks from which objects are constructed. Fields can have one of the following four data types:

- 32-bit integer
- Boolean
- Character string
- Enumeration value

Each field definition is uniquely identified by a field ID. A field can contain a single data element, or it can contain a list of data elements of the same type. The NetView for AIX program provides a number of routines to create and manipulate fields. Other API routines retrieve field values and field information from the object database in various ways. By themselves, fields are not very useful. Only after a field is associated with an object can you manipulate its data, such as setting or retrieving field values. Fields can contain data only when they are associated with objects.

Two name fields are predefined by the NetView for AIX program and are available for application use:

- Selection Name
- IP hostname (IP networks only)

Applications should not redefine or change these predefined NetView for AIX fields.

The field creation routines presented in this section enable you to create fields and retrieve field information from the object database; they do not enable you to define field values. “Creating Objects” on page 87 describes how to create objects and assign initial field values for fields in the object.

To create a field, use either the field registration file (FRF) or the OVwDbCreateField() routine. API routines are used to set field values. You can use existing fields or create new ones as your application needs dictate.

Using the Field Registration File

The field registration file (FRF) is the preferred way to create fields. The OVwDbCreateField() routine is available for flexibility, but should be used only in restricted cases. There are two important reasons why you should use the FRF to create fields:

1. The NetView for AIX program parses all application and symbol-type registration files when it is started. If an entry in these files refers to a field not already defined, the application may fail. Using the field registration file to define a field guarantees that a field definition exists when the NetView for AIX program is started.
2. The FRF displays field specifications to users and developers. Users can examine the set of field registration files to see which fields are present in the object database. If fields are created through the API, their existence can only be determined programmatically.

Upon installation, the English definitions of the fields files are stored in /usr/OV/fields/C. Any English field definitions that you decide to add should also be placed in the /usr/OV/fields/C directory. For non-English definitions, you can create a $LANG directory under /usr/OV/fields and add the new field files to that directory. If you do this, you must then link all the fields files from /usr/OV/fields/C to the new /usr/OV/fields/$LANG directory so that ovw will recognize them.

Whether you create a new directory or use /usr/OV/fields/C for your field registration files, do not put any files that are not field registration files into the directory. When ovw processes field registration files, it will fail if it encounters a file containing something other than field registration declarations, or if it encounters an empty subdirectory under the /usr/OV/fields directory.

After defining fields with field registration files, you can construct objects based on these fields. See “Creating Objects” on page 87 for more information about creating objects.

Processing Field Registration Files

Unlike ARFs and STRFs, FRFs are not processed automatically each time the NetView for AIX program is started. To tell the NetView for AIX program to read FRFs, issue the nv6000 command with the -fields option:

```
   nv6000 -fields
```

"How Registration Files Are Processed" on page 42 describes the processing performed for registration files. For more details, refer to the OVwRegIntro man page.

Building a Field Definition

A field definition consists of the following components:

- The field name
- The data type of the field
- Flags that indicate how the field is used

Specify these components in a Field block. Every field must have a data type and a unique name. Flags can be specified to indicate how the field is to be treated by the NetView for AIX program.

Each FRF contains one or more Field blocks. Each Field block defines a single field and has the following syntax:

```
Field "field name" {
   Type <field type>;
   [Enumeration <field enumeration>;]
   [Flags <field flags>;]
}
```

The field name and field type are required; the enumeration and flags entries are optional.

Field Type

Every field has an associated data type. The `Type` entry declares the field's data type. Type is a required entry, and must be set to one of the following types:

Boolean	This type can only have the values True or False.
Integer32	A 32-bit signed integer.
String	A standard character string, limited to 256 characters.
Enumeration	Declares the field to be an enumerated type. All possible values for the enumerated type are declared in an accompanying Enumeration entry. An example enumeration appears in "Example of Enumerated Types and Locate Flag" on page 83.

Field Flags

While the `Type` field specifies the data type of the field, the `Flags` field specifies how the field is treated. The NetView for AIX program gives you the flexibility to treat fields in different ways. For instance, you can specify that a particular field is a name field, which means that the value of the field is unique across all objects that contain the field. You can also specify that a particular field should appear in the `Locate..By Attribute` dialog box on the NetView for AIX menu interface. These behaviors, as well as others, are defined using the `Flags` statement. There are five types of flags (behavior) that can be applied to fields:

List Specifies that the field contains a list of fields of the same type. The list flag allows you to associate several values of the same type with a single field definition. The list flag is valid only for the string and integer data types. For example, you might define a field for the names of administrators of your computer network. The type is string, and the flag field is set to list to allow you to store more than one name.

Name Indicates that the value of the field uniquely identifies an object containing this field definition. This flag can be used only for fields of type string. Non-string data types with unique values, such as a network adapter's link-level address, can serve as name fields provided that they are converted to and stored as strings.

Locate By specifying this flag in the field definition, the field name shows up in the `Locate..By Attribute...` dialog box when users attempt to locate an object.

Use care in setting this field. If you do not specify the locate flag, users cannot locate an object based on your field. On the other hand, indiscriminate use of the locate field will result in an overabundance of locate entries, making that dialog box harder to use effectively. Set this flag only if users will need to locate objects through this field. This flag cannot be set together with the list flag.

Capability This field is used to classify an object. For example, the NetView for AIX program defines fields, such as isRouter and isDevice. These fields are used to classify objects and are often used in assertions. The capability flag may only be used for Boolean and enumerated field types. Capability fields determine menu greying (which menu operations are available based on the selected objects).

For an example of the use of Capability fields, see "Defining Rules within Enroll Blocks" on page 38.

General Fields with this flag appear in a special General Attributes dialog box associated with every object. This is for fields that are not application-specific and do not appear in any application-specific dialog box. The vendor field is a good example of a general field.

Combining Data Types and Flags

As you can see from the flag descriptions, not all flags can be used with all data types. Table 7 summarizes valid data type and flag combinations.

Table 7. Valid Data Type and Flag Combinations

Flag	**Data Type**			
	Integer	**String**	**Boolean**	**Enumeration**
List	●	●		
Name		●		
Capability			●	●
Locate	●	●	●	●
General	●	●	●	●

Combinations of Flags

Flags can be applied to different data types and for different purposes. There may be occasions where you want to use flags in combination. Flags can be used in the following ways:

- Any flag can be used individually.
- The `name` and `locate` flags can be used together.
- The `capability` and `locate` flags can be used together.
- The `general` flag can be used with the other valid combinations described here.

The combinations listed here are the only valid combinations. For example, you cannot combine the `name` and `capability` flags in a single flags statement.

Examples of Field Definitions

This section shows how to use field types and flags.

Example of Capability Fields

The three fields shown in the following examples are used to determine an object's capabilities. An object could have device, computer, or node capabilities. In fact, an object can have a combination of these capabilities. Capability fields are used to classify objects in the NetView for AIX program; therefore, capability fields cannot be used to uniquely identify (or name) an object.

```
Field "isNode" {
    Type boolean;
    Flags capability;
}

Field "isDevice" {
    Type boolean;
    Flags capability;
}

Field "isComputer" {
    Type boolean;
    Flags capability;
}
```

Example of Enumerated Types and Locate Flag

This example shows how you can use enumerated types to classify the status of nodes on a network. Each node on a network has a status value from this set. By setting the locate flag, users have the ability to search for, or locate, all nodes on the network with a particular status. You should not set the capability flag or name flag for this field.

```
Field "IP status" {
    Type enumeration;
    Enumeration "Unset",
                "Unknown",
                "Normal",
                "Marginal",
                "Critical"
                "Unmanaged"
                "Acknowledged"
                "UserStatus1"
                "UserStatus2";
    Flags locate;
}
```

Example of a Name Flag

Because the hostname of a computer is unique for all nodes on the network, the name flag can be set for the field.

```
Field "IP Hostname" {
    Type string;
    Flags name;
}
```

Example of a List Flag

The following example demonstrates how to use the list flag in field definitions. This field definition specifies that the field's value is a list of strings, such as the names of people who can service equipment. Values are added to lists using the OVwDbSetFieldValue() routine.

```
Field "Service Contacts" {
    Type string;
    Flags list;
}
```

Using the OVwDbCreateField() Routine

Fields are created using the OVwDbCreateField() routine. This routine creates an entry in the NetView for AIX database that describes how data will be stored and treated. This routine does not assign data for the field (the field value). You can assign a field value only after you have associated that field with an object. This process is described in “Creating Objects” on page 87. Once an object exists, you are free to manipulate all field values associated with that particular object.

Though the same field can be used in different objects, each object can have a different field value. The OVwDbCreateField() call has the following format:

```
OVwFieldId OVwDbCreateField( char *fieldName,
                             int fieldType, unsigned int fieldFlags );
```

When calling OVwDbCreateField, you need to supply these arguments:

- The `fieldName` argument is a character string containing the textual name for the field.
- The `fieldType` argument is the data type. Valid types are `ovwIntField`, `ovwBooleanField`, `ovwStringField`, and `ovwEnumField`. If the data type is a list, then the `fieldFlags` argument will have the `ovwListField` value set.

- The `fieldFlags` argument is a logical expression containing the flags to be set for the new field. Valid flags are `ovwListField`, `ovwNameField`, `ovwCapabilityField`, `ovwLocateField`, `and ovwGeneralField`. (Valid combinations of field flags were described in “Combinations of Flags” on page 83.) The `fieldFlags` argument is zero (0) if no flags are needed. Valid types and flags are defined in `<OV/ovw_obj.h>`.

The OVwDbCreateField() call returns an identifier of type OVwFieldId. This identifier is used in subsequent calls involving the field. At this point, the field exists in the database, and values can be set for the field for particular objects. The following short code segment shows how to create a field that is a list of integers.

```
 ...
OVwFieldId field_id;
field_id = OVwDbCreateField( "a_test", ovwIntField, ovwListField );
```

Identifying a Field

Most API routines use the field ID, not the field name, when dealing with fields. Both are unique ways to identify a field. You are free to use either form in your programs, as long as it is converted to the appropriate form for calls to API routines. Two NetView for AIX API routines are available to convert between field names and field IDs. They have the following formats:

```
OVwFieldId OVwDbFieldNameToFieldId( char *fieldName );
char *OVwDbFieldIdToFieldName( OVwFieldId fieldId );
```

Applications that create fields using the field registration file will need to use the OVwDbFieldNameToFieldId() routine at some time. To use routines that take a field ID as input, you will need first to convert the field name in the field registration file to a field ID using OVwDbFieldNameToFieldId(). Refer to the man pages if you need more information about these routines.

Defining Enumeration Values

The NetView for AIX program supports enumerated types as one of the field data types. If you use enumerated types, and do not use an FRF to define the field, you must follow the procedures in this section to define the values for the enumeration.

1. Allocate memory for the appropriate data structures. You must allocate memory for two data structures:
   - An array of pointers, where each array entry points to one of the enumeration value strings.
   - The OVwEnumConstants data structure contains a count of the number of enumeration values and a pointer to the array containing pointers to the enumeration values. The data structure has the following format:

     ```
         typedef
     struct {
            int count;
          char **names;
        } OVwEnumConstants;
     ```
2. Define each enumeration value. Store the address of the string in the array of string pointers.
3. Call the OVwDbSetEnumConstants() routine to set the values.

The following code segment shows how to define an enumerated type with three possible values:

```
 ...
OVwFieldId id;
OVwEnumConstants *p;
int ret;

/* create the field without special flags */
id = OVwDbCreateField( "enum_test", ovwEnumField, 0 );
if ( OVwIsIdNull( id ) ) {
/* error processing */
}

/* allocate the memory for the enumerated constants
structure */
p = (OVwEnumConstants *) malloc(sizeof(OVwEnumConstants));

/* allocate the memory for the pointers to the 3 names */
p->names = (char **) malloc(sizeof(char *) * 3);

/* now assign each name */
p->count = 3;
p->names[0] = "red";
p->names[1] = "green";
p->names[2] = "blue";

ret = OVwDbSetEnumConstants( id, p );
```

Retrieving Field Information

The NetView for AIX program provides routines that return information about a field. These routines return such information as the field ID, the field name, and the field data type. They return information about the field, rather than field values for particular objects. You can retrieve information for a single field or for all fields with particular characteristics in the object database.

Use the OVwFieldInfo data structure to retrieve information about a field. The data structure contains the field name, type, and flags, as well as the field ID. The data structure has the following format:

```
typedef struct {
   OVwFieldId field_id;
   char *field_name;
   int field_type;
   unsigned int field_flags;
} OVwFieldInfo;
```

For a Single Field

The OVwDbGetFieldInfo() routine returns field information for a single field. When called with a field ID, it returns a pointer to an OVwDbFieldInfo data structure containing field information. Because the memory for the structure is dynamically allocated by the NetView for AIX program, you should use the associated API routine to free the memory when the application is finished. The following code segment illustrates:

```
OVwFieldInfo *fp;
OVwFieldId id;
 ...
/* Create a boolean field */
id = OVwDbCreateField( "test", ovwBooleanField, 0 );
 ...
/* Retrieve the field information from the database */
fp = OVwDbGetFieldInfo( id );
printf( "field_id    is %d\n, ", fp->field_id    );
printf( "field_name  is %s\n, ", fp->field_name  );
printf( "field_type  is %d\n, ", fp->field_type  );
printf( "field_flags is 0x%x\n", fp->field_flags );
OVwDbFreeFieldInfo( fp );
```

For All Fields

The OVwListFields() routine searches the entire NetView for AIX object database and returns a list of field information for fields that have the appropriate flags set. Valid flags are `ovwAllFields`, `ovwNameField`, `ovwLocateField`, `ovwCapabilityField`, and `ovwGeneralField`. For example, the following code segment shows how to retrieve a list of all fields in the object database that have the `ovwNameField` flag set:

```
OVwFieldList *flp;
OVwFieldInfo *fip;
int i;
 ...
flp = OVwDbListFields( ovwNameField );
for ( i=0, fip=flp->field_list; i<count; i++, fip++ ) {
   <process the entry pointed to by fip>
}
OVwDbFreeFieldList( flp );
```

Getting Enumeration Type Values

After you have defined an enumeration, you can use a number of API routines to retrieve information about the enumeration. You can retrieve the symbolic name list with the OVwDbGetEnumConstants() routine and translate between symbol names and values with the OVwDbGetEnumValue() and OVwDbGetEnumName() routines. Refer to the man pages for more information on these and other related routines that operate on enumerated data types.

Creating Objects

In the NetView for AIX program, an object is an internal representation of a logical or physical entity or resource that exists in a computer network. An NetView for AIX object consists of a unique object identifier and a set of fields that specify all the characteristics of the object. The NetView for AIX program provides routines that create and delete objects, as well as routines that change the object fields. Other NetView for AIX routines perform such functions as retrieving the list of all fields associated with an object and searching the object database for objects that contain specific field values.

When you create an object, an object definition is added to the NetView for AIX object database. The object definition is created with a selection name and an optional field value.

The OVwFieldValue Data Structure

The OVwFieldValue data structure defines field values. Most NetView for AIX object database routines use the OVwFieldValue data structures in some form, either as an argument or a returned result. The field value structure is implemented as follows:

```
typedef struct {
   OVwBoolean is_list;
   int field_type;
   union {
      int32 int_val;
      OVwBoolean bool_val;
      char *string_val;
      int enum_val;
      OVwListFieldValue *list_val;
   } un;
   OVwBoolean modified;
} OVwFieldValue;
```

When used for simple data types, the `is_list` boolean value is set to FALSE, and the field value is stored as an integer, boolean, string, or enumerated value. The `field_type` entry specifies which union entry is used. The `modified` entry is not used during field creation and can be temporarily ignored. It is used during dialog-box processing to indicate whether a field has been modified by the user and thus must be processed by the application that handles entries made through that dialog box.

If the field value is a list, then the `is_list` entry is set to TRUE and `list_val` points to a structure that heads the list. The following additional data structures define the list head and the list entries:

```
typedef struct {
   union {
      int32 int_val;
      OVwBoolean bool_val;
      char *string_val;
      int enum_val;
   } un;
   OVwBoolean selected;
} OVwListFieldEntry;

typedef struct {
   int count;
   OVwListFieldEntry *list;
} OVwListFieldValue;
```

Note that for lists the data type is not specified in the OVwListFieldEntry data structure. Because all items in the list must be the same type, it is sufficient to specify the data type in the OVwFieldValue data structure. The selected entry in the OVwListFieldEntry can be temporarily ignored.

Note that the list of values is not implemented as a standard linked list. The OVwListFieldValue structure that heads the list contains an integer count of the number of list entries, followed by a pointer to a contiguous array of list entries. For internal memory management reasons, most list data structures are implemented in this way.

Selection Name

A selection name is a special name field that uniquely identifies an object and is set for all objects. Before describing how the selection name is used, we need to review some name-field concepts.

The purpose of the selection name is to make sure that every object has a textual name that can be displayed through the user interface to identify the object. The selection name is the principal name by which the object is known through the user interface. Because the selection name is intended for presentation purposes and can be changed by a user, applications should not be designed to rely on the value of the selection name.

Every object is uniquely identified by an object ID that is returned when the object is created. In addition, an object can have many name fields, such as a hostname for TCP/IP networking, a Fully Distinguished Name for OSI networking, or a name created by an application. A name-field value for any object is unique. Therefore, any name field can be used to uniquely identify an object.

A selection name is required for every NetView for AIX object regardless of other name fields the object might have. You can define your own selection name value when your object is created, or you can let the NetView for AIX program choose a value for you.

Creating Objects with API Routines

Three API routines are available to create objects:

- OVwDbCreateObject(), a generic routine that can create any type of NetView for AIX object
- OVwDbCreateObjectByHostname(), which creates an object by its hostname
- OVwDbCreateObjectBySelectionName(), which creates an object by its selection name

In general, objects are created with the OVwDbCreateObject() routine. It has the following format:

```
OVwObjectId OVwDbCreateObject( OVwFieldBinding *name );
```

The `name` argument is a pointer to an OVwFieldBinding structure, which is simply a structure that links a field ID with a field value. The OVwFieldBinding structure is defined as follows:

```
typedef struct {
   OVwFieldId     field_id;
   OVwFieldValue *field_val;
} OVwFieldBinding;
```

If you call OVwDbCreateObject() with a NULL parameter, the NetView for AIX program will create an object for you and assign it a system-generated selection name. If you supply a pointer to a valid OVwFieldBinding structure in the OVwDbCreateObject() routine call, one of three things can happen:

- If the field value you supply is a selection name, the NetView for AIX program will attempt to define the object. If the selection name is already allocated, the call returns an error. You can optionally use the OVwDbGetUniqObjectName() routine to generate a selection name that is guaranteed to be unique. The

OVwDbGetUniqObjectName() routine is described in “Generating Unique Name Field Values” on page 90.

- If the field value is a name field other than the selection name, the NetView for AIX program will attempt to define the object. If the name-field value is not unique, an error will be returned. If the name is unique, the NetView for AIX program will define the object, set the name-field value, and generate a selection name for you based on the name field you supply.
- If the field value you provide is not a name field, the NetView for AIX program will create a selection name for you. The name will be in the form `Selection Name`*n*, where *n* is a unique integer. You can change the selection name later if you desire.

The following example shows how to create an object that contains a name field. In this example, the name field is the NetBIOS system name, as defined in a field registration file. Since the field is not a selection name, the NetView for AIX program will also generate a selection name for the object:

```
OVwFieldId f_id;
OVwObjectId obj_id;
OVwFieldBinding *bp;

/* get the field ID associated with the NetBIOS Name */
f_id = OVwDbFieldNameToFieldId( "NetBIOS Name" )
if ( OVwIsIdNull( f_id ) ) {
   /* error processing */
}

/* allocate an OVwFieldBinding structure */
bp = (OVwFieldBinding *) malloc( sizeof(OVwFieldBinding) );

/* fill in the field_id entry and the field_value structure */
bp->field_id = f_id;

bp->field_val = (OVwFieldValue *) malloc( sizeof(OVwFieldValue) );
bp->field_val->un.string_val = "nametest";
bp->field_val->field_type = ovwStringField;
bp->field_val->is_list = FALSE;

/* create the object and save the object ID that is returned */
obj_id = OVwDbCreateObject( bp );
if ( OVwIsIdNull( obj_id ) ) {
   /* error processing */
}
 ...
```

Generating Unique Name Field Values

The OVwDbCreateObject() routine returns an error if you attempt to create an object with a name field value that is not unique. You can generate a new name and try the call again, but there is no guarantee that your new name is unique. The NetView for AIX program solves this problem by providing a routine that generates a unique name for you, using a name you provide as a base. The OVwDbGetUniqObjectName routine has the following function prototype:

```
char *OVwDbGetUniqObjectName( OVwFieldId nameFieldId,
                              char *nameValue );
```

Here is how the routine behaves:

- If you provide a name-field value that is not already assigned, the same name is returned to you.
- If you provide a name-field value that is already assigned, the NetView for AIX program will choose a unique name based on the value you provide. The NetView for AIX program will append an integer to the base name, thereby making the newly formed name unique.
- If you pass a NULL parameter for the `nameValue` parameter, the NetView for AIX program will choose a unique name-field value for you. The NetView for AIX program will convert the field ID to a base field name, and will append an integer to the base field name to make the new name unique.

The name generated by the call will be unique, which lets you call the OVwDbCreateField() routine with assurance that the field name is not already allocated.

Creating Objects by Host Name or Selection Name

You can use the general purpose OVwDbCreateObject() routine to create an object that has a hostname or selection name. The following steps are involved:

1. Call the OVwDbFieldNameToFieldId() routine to retrieve the field ID associated with the Hostname or Selection Name. (The "IP Hostname" and "Selection Name" fields are already present in the database, so you don't need to create them.)
2. Allocate memory for the OVwFieldValue structure.
3. Fill in the OVwFieldValue structure, setting the `string_val` entry to the name.
4. Allocate memory for an OVwFieldBinding structure, set the field ID, and set a pointer to the field value.
5. Call the OVwDbCreateObject() routine.

As a convenience, the NetView for AIX program provides two routines that create objects by IP Hostname or Selection Name. They require only a single call, and they have the following formats:

```
OVwObjectId OVwDbCreateObjectByHostname( char *hostname );
OVwObjectId OVwDbCreateObjectBySelectionName( char *selectionName );
```

These routines are a convenient alternative to using the generic OVwDbCreateObject() routine.

Getting and Setting Object Field Values

After creating fields and objects, you can use the remaining object database routines to manipulate the fields in a wide variety of ways. The NetView for AIX object database routines described in this section enable you to:

- Get and set field values in the NetView for AIX object database.
- Associate new fields with existing objects.
- Retrieve a list of all the fields associated with an object.
- Retrieve the list of capability or name fields set for an object.
- Get all objects in the object database that contain specific field values.
- Convert between any field name and field ID.

Using Basic Routines to Access Field and Object Values

Using the basic routines presented in this section, you can access all field and object values in the NetView for AIX object database. Many convenience versions of these basic calls are also available. The convenience versions reduce the number of parameters or simplify the result returned. In some cases, the convenience versions reduce the number of calls you need to make to get field information. However, they do not perform actions that cannot be performed with the basic calls.

The most basic way to retrieve field value information is with the OVwDbGetFieldValue() routine. Given a field ID and object ID, this routine returns a pointer to a field value structure. The OVwDbGetFieldValue() routine has the following function prototype:

```
OVwFieldValue *OVwDbGetFieldValue( OVwObjectId objectId,
                                   OVwFieldId fieldId );
```

Because the memory used to store the returned field value is allocated by the NetView for AIX program, you should call the OVwDbFreeFieldValue() routine to free the memory that is no longer needed.

The following example shows how to retrieve the selection name-field value from the NetView for AIX object database. This example converts the field name to a field ID and calls OVwDbGetFieldValue() to retrieve the field value information.

```
OVwFieldId field_id;
OVwObjectId obj_id;
OVwFieldValue *val_ptr;


 ...
field_id = OVwDbFieldNameToFieldId( ovwNselectionName );

/* Get the field value.  Assume that 'obj_id' is set for our object. */
val_ptr = OVwDbGetFieldValue( obj_id, field_id );
printf( "The Selection Name for this object is %s\n",
        val_ptr->un.string_val );
OVwDbFreeFieldValue( val_ptr );
 ...
```

Note that the `ovwNselectionName` string constant is passed in the call to OVwDbFieldNameToFieldId(). Using this constant is less prone to programming error than specifying the selection name. The file `<OV/ovw_fields.h>`, which contains string constant definitions, is included for you automatically if you include `<OV/ovw.h>`.

Setting Field Values

The OVwDbSetFieldValue() routine is used to set field values as well as to add fields to existing objects. It has the following format:

```
int OVwDbSetFieldValue( OVwObjectId objectId,
                        OVwFieldId fieldId,
                        OVwFieldValue *fieldValue );
```

To set a field value, you must allocate memory for an OVwFieldValue data structure, set the appropriate entries, and call the OVwDbSetFieldValue() routine. This technique was shown in “Creating Objects” on page 87. If you attempt to set a field value for a field that is not already associated to the object, the NetView for

AIX program will add the field to the object. Note that the OVwDbCreateObject() routine is used to set a single field value and to create an object. The OVwDbSetFieldValue() routine is used to add other fields to the object and set their value.

Using Convenience Routines

Because retrieving and setting field values are common operations, the NetView for AIX program provides convenience routines that simplify these tasks. Some convenience routines retrieve particular field values directly, bypassing the overhead of using the OVwFieldValue data structure upon return. Examples include OVwDbGetFieldIntegerValue() and OVwDbGetFieldBooleanValue(). To use these routines, you must know the data type of the field. Other convenience routines eliminate the need to set up an OVwFieldValue data structure. The desired value is passed as an argument to the appropriate routine. Examples of these routine types include OVwDbSetFieldStringValue() and OVwDbSetSelectionName().

Table 8 lists some of the NetView for AIX program's object database convenience routines.

Table 8. NetView for AIX Object Database Convenience Routines

Convenience Routine	Purpose

OVwDbGetFieldIntegerValue()
OVwDbGetFieldBooleanValue()
OVwDbGetFieldStringValue()
OVwDbGetFieldEnumByValue()
OVwDbGetFieldEnumByName()
	These routines take an object ID and field ID as arguments and return the appropriate field value. They save you the trouble of having to dereference pointers into the OVwFieldValue structure.

OVwDbSetFieldIntegerValue()
OVwDbSetFieldBooleanValue()
OVwDbSetFieldStringValue()
OVwDbSetFieldEnumByValue()
OVwDbSetFieldEnumByName()
	These routines take an object ID, field ID, and value as arguments and set the appropriate field value. You do not need to set up the OVwFieldValue structure if you use these routines.

OVwDbSetSelectionName()
OVwDbSetHostname()
	These routines use an object ID and a string to set the appropriate name field.

OVwDbGetCapabilityFieldValues()
OVwDbGetNameFieldValues()
	These routines return a list of name or capability fields set for the object.

OVwDbNameToObjectId()	This routine converts the value of any name field to the object ID it identifies.

OVwDbSelectionNameToObjectId()
OVwDbObjectIdToSelectionName()
OVwDbHostnameToObjectId()
OVwDbObjectIdToHostname()
	These routines translate between selection name, or object ID and hostname, or object ID.

OVwDbListObjectsByFieldValue()
OVwDbListObjectsByFieldValues()
OVwDbFreeObjectIdList()
	The Locate calls search the entire NetView for AIX object database for objects and fields that match certain criteria. Because they search the entire database, they are slow.

OVwDbGetFieldValuesByObjects()
OVwDbFreeObjectFieldList()
	These routines get a list of values for a certain field for a list of objects.

These routines are not described here, because their use is straightforward after the basic calls are understood. Refer to the man pages for more information about these routines.

Getting a List of Object Fields

The NetView for AIX program provides routines that can return a list containing several fields, each of which may have different types. For example, the OVwDbGetFieldValues() routine returns a list of all field values for fields associated with an object. The OVwDbGetFieldValues() routine returns the full definition of a particular object, because it returns all the fields of an object.

Representing lists of fields with different types requires a new data structure. The OVwFieldBindList data structure performs this function and has the following format:

```
typedef struct {
   int count;
   OVwFieldBinding *fields;
} OVwFieldBindList;
```

This structure follows the list scheme used by other NetView for AIX list data structures. The *fields* variable points to the first entry in a contiguous array (of size count) of data structures (in this case, OVwFieldBinding data structures).

Figure 2 illustrates the OVwFieldBindList structure.

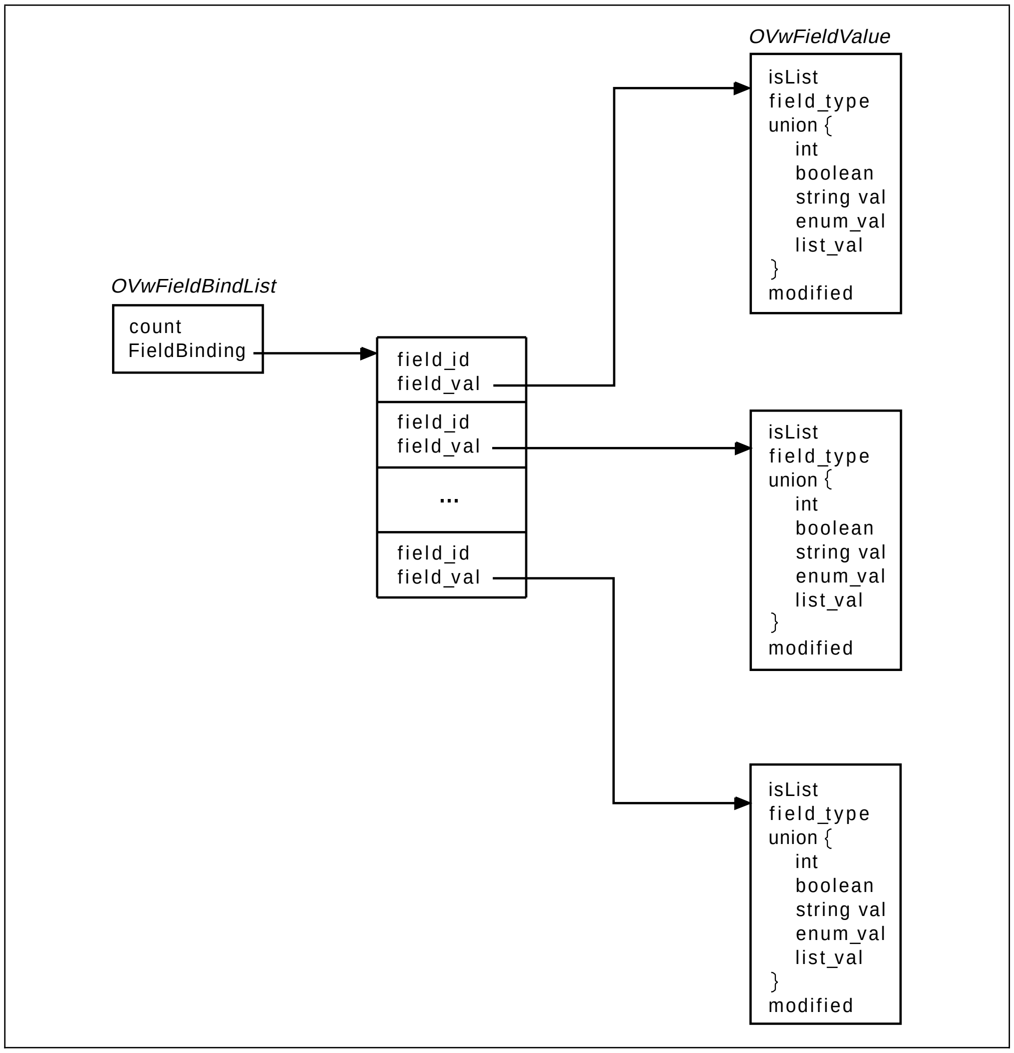

Figure 2. Field Binding List Data Structure

The following example shows how the OVwFieldBindList data structure is used. For convenience, assume that each field is a simple type, not a list of simple types.

```
OVwFieldBindList *fbl_ptr;
OVwFieldBinding *fb_ptr;
OVwObjectId obj_id;
int i;

 ..
/* assume that 'obj_id' is the object ID for our object */

fbl_ptr = OVwDbGetFieldValues( obj_id );
for ( i=0, fb_ptr=fbl_ptr->fields; i<fbl_ptr->count; i++,
fb_ptr++ ) {
   printf( "Processing field binding entry %d\n", i );
   printf( "   The field ID is %d\n", fb_ptr->field_id );
   switch ( fb_ptr->field_val->field_type ) {
      case ovwStringField:
          printf( "   String value is %s\n",
                  fb_ptr->field_val->un.string_val );
          break;
      case ovwIntField:
          printf( "   Integer value is %d\n",
                  fb_ptr->field_val->un.int_val );
          break;
      case ovwBooleanField:
          printf( "   Boolean value is %d\n",
                  fb_ptr->field_val->un.bool_val );
          break;
      case ovwEnumField:
          printf( "   Enum value is %d\n",
                  fb_ptr->field_val->un.enum_val );
          break;
   }
}
OVwDbFreeFieldBindList( fbl_ptr );
 ...
```

Retrieving Object Attribute Information

Some object attributes are maintained globally, and some apply only to the map on which that object appears. Generally, developers do not need to be concerned with whether an object attribute is global or map-specific. When setting attributes, the NetView for AIX API routines take care of setting the appropriate global or map-specific object attributes.

When an application sets status for an object, the status is set only for the object on the open map. Setting status on that object does not affect the status of the same object on another map. Object status is an example of an attribute that is map-specific.

There are a number of other map-specific object attributes as well. These map-specific object attributes, and the routines that operate on them, are described in this section. For example, when calling the OVwSetStatusOnObjects() routine, the NetView for AIX program automatically sets the map-specific object attribute.

When retrieving object attributes, note the distinction between global and map-specific attributes. If the application needs global attribute information, the developer should use the OVwDbGetFieldValues() or related routine. Those routines were described earlier in this chapter. When retrieving map-specific information, the developer should use the OVwGetObjectInfo() routine or another routine presented in this section.

The OVwObjectInfo Data Structure

The OVwObjectInfo data structure stores map-specific object information. This data structure is used by NetView for AIX routines that retrieve map-specific object information. The OVwObjectInfo data structure has the following definition:

```
typedef struct {
   OVwObjectId object_id;          /* object ID */
   OVwSubmapId child_submap_id;    /* child submap of object */
   int num_symbols;                /* number of symbols on open map */
   OVwStatusType object_status;    /* object-specific status */
   OVwStatusType compound_status;  /* compound status */
   int op_scope;                   /* ovwNotApplicable,
                                      ovwOpenMapScope, ... */
   OVwFieldBindList *field_values; /* object capability field values */
} OVwObjectInfo;
```

Note that all of the fields in this structure apply to the object only as it relates to the current map. They do not apply to instances of the object on other maps. Most of the fields in the data structure are self-explanatory; however, two fields require further description.

The `op_scope` and `field_values` fields have special uses that are not apparent from their names. Both fields are used to provide particular information to callback routines when specific events occur.

- The `op_scope` field indicates the scope of delete operations.
- The `field_values` field is used to supply certain object fields to an application callback routine for `ovwConfirmCapabilityChange` and `ovwConfirmDeleteObjects` events.

The `<OV/ovw.h>` include file contains the definition for the OVwObjectInfo data structure, as well as additional comments about the fields and how they are used.

The OVwGetObjectInfo() Routine

The OVwGetObjectInfo() routine retrieves map-specific object information for any object. Given an object ID, this routine returns a pointer to an OVwObjectInfo data structure containing complete map-specific object information. An associated OVwFreeObjectInfo() API routine frees memory allocated by the OVwGetObjectInfo() routine. They have the following formats:

```
OVwObjectInfo *OVwGetObjectInfo( OVwMapInfo *mapInfo,
                                 OVwObjectId objectId );
void OVwFreeObjectInfo( OVwObjectInfo *object );
```

Note that OVwGetObjectInfo() returns only the map-specific information relating to an object. If you need global object information, use the OVwDbGetFieldValues() routine.

Getting Symbol IDs Associated with an Object

The OVwGetSymbolsByObject() routine provides a list of all the symbols that represent a given object on the open map. For instance, consider the use of a selection list in a callback routine. When the NetView for AIX program invokes a callback routine, it supplies a copy of the current selection list. The selection list is a list of object IDs. Use the OVwGetSymbolsByObject() routine to convert the list of object IDs to a list of all symbols that represent the given object.

The OVwGetSymbolsByObject() routine has the following function prototype:

```
OVwSymbolList *OVwGetSymbolsByObject(OVwMapInfo *mapInfo,
                                     OVwObjectId objectId );
```

This routine returns a pointer to a standard NetView for AIX list structure, composed of an integer count of the number of entries in the list and a pointer to the first in a contiguous array of list entries. Be sure to free the dynamically allocated memory with the OVwFreeSymbolList() routine when you are finished.

Getting a List of All Objects on a Map

The OVwListObjectsOnMap() routine is useful for getting a list of all the objects that are present on the open map. The OVwListObjectsOnMap() routine has the following function prototype:

```
OVwObjectList *OVwListObjectsOnMap( OVwMapInfo *mapInfo,
                                    OVwFieldBindList *fieldValues );
```

The OVwListObjectsOnMap() routine returns a pointer to an OVwObjectList structure. The OVwObjectList structure is another standard NetView for AIX list structure that is composed of an integer count of the number of entries in the list and a pointer to the first in a contiguous array of list entries. The `fieldValues` argument is an optional argument that the NetView for AIX program uses to filter the list of objects returned by the OVwListObjectsOnMap() routine. If `field_values` is specified, the list of returned objects is filtered to include only those objects that have the specified field values. If `field_values` is NULL, the NetView for AIX program returns a list of all objects in the open map.

Call the OVwFreeObjectList() routine to free the dynamically allocated memory when you are finished.

Deleting Objects

Objects should be deleted from the NetView for AIX object database when they are no longer needed. The NetView for AIX program's process for deleting an object addresses the possibility that the object might still be used by another application. The NetView for AIX program provides a set of routines that applications can call to cooperatively delete an object. This process guarantees that an object is not deleted while it is being used by another application.

An object should be deleted when it is deleted from the last submap on which it appears. Applications can determine that an object is no longer in use by registering for the `ovwConfirmDeleteObjects` event. See Chapter 10, “Map Events and Map Editing” on page 145 for more information about map editing and map events. The OVwConfirmDeleteObjectsCB() man page also describes when to delete objects.

When an application has finished using an object, the application should be designed to:

1. Delete all symbols that represent the object. See “Deleting Symbols” on page 129 for information on deleting symbols.
2. Call OVwDbUnsetFieldValue() for every field that it controls or sets. The OVwDbUnsetFieldValue() routine has the following format:

   ```
   int OVwDbUnsetFieldValue( OVwObjectId objectId, OVwFieldId fieldId );
   ```

3. Call the OVwDbDeleteObject() routine to delete the object. The OVwDbDeleteObject() routine has the function prototype:

   ```
   int OVwDbDeleteObject( OVwObjectId objId );
   ```

   The NetView for AIX program will delete the object if either of the following conditions exists:

   - No fields are set for the object.
   - The object only has fields with either the capability or general flag set.

If either of these conditions is true, the NetView for AIX program will delete the object. If any application has fields set for the object, the object will not be deleted until that application unsets its field values and calls the OVwDbDeleteObject() routine. Note also that before deleting an object, you must delete all the symbols that represent that object from all maps. Refer to the OVwDbUnsetFieldValue() and OVwDbDeleteObject() man pages for more information about deleting objects.

Chapter 8. Creating and Using Symbols

This chapter describes ways to define and use symbols with the NetView for AIX program. Read this chapter if your application deals with symbols on NetView for AIX submaps. You should be familiar with NetView for AIX objects, maps, and submaps.

Topics in this chapter include:

- Symbol attributes
- Creating symbols
- Changing symbol appearance and behavior
- Retrieving symbol information
- Deleting symbols

This section reviews symbol concepts that you must know before you can create and use symbols.

The terms *symbol* and *object* appear throughout this chapter. You must understand the distinction between symbols and objects before you can use the NetView for AIX symbol routines. A symbol is a graphical representation of an object as it appears on a submap of a particular map. Symbols are presentation elements, while objects are underlying database elements that represent network elements. An object can be represented by multiple symbols.

Understanding Symbol Attributes

Symbols have attributes beyond those of the objects they represent. Important symbol attributes include:

- Label
- Status
- Status source
- Symbol type
- Plane location
- Position
- Behavior

These attributes can vary among the different symbols representing a particular object, because they are symbol-specific rather than object-specific. This section describes each of these symbol characteristics.

Symbol Label

The symbol label is a text string that is displayed at the bottom of each symbol. The symbol label is provided as a convenience to users. It does not have to be unique, because it is not used to uniquely identify the symbol. (Numeric symbol IDs uniquely identify symbols.) An application can assign the initial value of the symbol label when the symbol is created, but users can change the label. Applications or users can enable or disable the display of the label.

Symbol Type

The symbol type determines the graphical representation of a symbol. The term *symbol type* is a convenient way to refer to the symbol class and subclass that define how the symbol is displayed. The symbol class defines the symbol category, and the symbol subclass defines a particular element within that class. The symbol class defines the outline, or shape and size, of the symbol; the subclass indicates the bitmap that is superimposed on that outline.

Symbol types can be defined with certain default capability fields. If a symbol with default capability fields is created in a submap, those capability fields will be added to the existing capability fields of the underlying object. Symbol classes, symbol subclasses, and default capability fields are described in detail in “Defining Symbols with Symbol Type Registration Files” on page 109.

Symbol Variety

There are two kinds of NetView for AIX symbols: icon symbols and connection symbols.

Icon symbols are displayed as a symbol graphic (or bitmap) within an outer shape, such as a circle, square, or diamond. In addition to the basic symbol attributes (label, status, status source, plane), an icon symbol has the following additional attributes:

- Class shape, based on the class of the symbol type
- Symbol graphic (or bitmap), based on the subclass of the symbol type
- Position

A connection symbol is a symbol that connects two icon symbols. A connection symbol appears as a line drawn between two icon symbols and has the following attributes beyond the basic symbol attributes:

- Line style based on the subclass of the symbol type
- Two connection end points

A connection symbol can connect any two icon symbols on the same submap. If the submap has a ring or bus layout, a connection symbol can also connect an icon symbol and the backbone symbol. A connection symbol cannot connect other connection symbols.

The variety of a symbol type is determined by its symbol class. For example, symbol types of the Computer class are of the icon variety, while symbol types of the Connection class are of the connection variety. “Changing a Symbol's Type” on page 122 explains how to programmatically change the symbol type of existing symbols. This is permitted as long as the new symbol type has the same variety as the old symbol. For example, you can change the symbol type of an icon symbol to a new icon symbol type, but not to a connection symbol type. Symbol variety is fixed for the life of the symbol.

Symbol Behavior

A symbol can either be *explodable* or *executable*. This characteristic determines what will happen when the user double-clicks on the symbol. Double-clicking on an explodable symbol results in the display of the child submap that shows the contents of the object represented by the symbol. Double-clicking on an executable symbol results in the invocation of an action provided by an application.

By default, symbols added to a map are explodable in nature. Both applications and users can change the behavior of a symbol. You can change the behavior of a symbol with the OVwSetSymbolBehavior() routine, which is described in “Changing a Symbol's Behavior” on page 123. You can define the selection list when you make the symbol executable.

Symbol Status

Each symbol displayed by the NetView for AIX program can display status information through the use of color. Each of the status values has an associated color that indicates the status of the symbol. Table 9 summarizes the status values, when they are used, and the color associated with each status condition:

Table 9. NetView for AIX Status Colors

Status	**Status Meaning**	**Icon Color**	**Connection Color**
Unknown	An application sets status to unknown when the status of an object cannot be determined.	Blue	Black
Normal	An application sets status to normal when the object is in a normal operational state.	Green	Black
Marginal	An application sets status to marginal when the operation of an object is impaired but still functional.	Yellow	Yellow
Critical	An application sets status to critical when an object is not functioning.	Red	Red
Unmanaged	Users can set this value. It indicates that the object should not be monitored and that status should be ignored.	Wheat	Black
Acknowledged	Users can set this value. It indicates that the object should not be monitored and that status should be ignored.	Green	Black
UserStatus1	This value can be defined by your administrator. See the *NetView for AIX User's Guide for Beginners* for more information.	Pink	Black
UserStatus2	This value can be defined by your administrator. See the *NetView for AIX User's Guide for Beginners* for more information.	Violet	Black

Although these status colors are preset by the NetView for AIX program, application developers or users can change them through the X resource file, /usr/OV/app-defaults/OVw.

Symbols can receive status from one of three sources:

Status by Object
This status source causes the symbol to reflect the status for the underlying object. It allows multiple symbol instances of the object to receive and reflect the same status information to the user.

It is recommended that you use this status source for symbols representing objects at the lowest level in the object hierarchy. Objects at this level typically do not have component objects below them. Examples of these types of objects include interface cards or specific software applications. This status source benefits users who copy symbol instances between submaps, because each new symbol instance has the same status as the original symbol instance.

Compound Status
This status source can be used for a symbol whose underlying object serves as a parent object for a child submap. Using this status source, the symbol status represents a summation of the status for all symbols in the child submap. This status source lets higher-level symbols reflect the state of lower level components controlled by multiple applications. The NetView for AIX program uses its own algorithm to determine how to summarize the status of symbols in a submap, but users can tune the algorithm to fit their needs. Developers have no control over the algorithm used to summarize status for a child submap. Compound status is appropriate only for symbols whose underlying objects serve as parent objects of child submaps.

Status by Symbol
This status source lets a specific symbol instance receive status directly from an application. Use this status source if your application has the exclusive responsibility for setting status for the specific symbol instance. Creating a new symbol instance with symbol status source assures application control over the symbol status. It also eliminates the possibility of status conflict between applications.

The *NetView for AIX Application Interface Style Guide* describes the effects of selecting a particular status mechanism. See that manual for more guidance in using status consistently across applications.

Symbol Position

The submap layout algorithm controls how symbols are placed on a submap. The layout algorithm is chosen during submap creation and is fixed for the life of the submap. Submaps can use any of the following layout algorithms:

- No Layout Algorithm
- Point to Point
- Bus
- Star
- Tree
- Ring
- Row/Column
- Multiple Connections

See “Submap Layout Algorithms” on page 135 for more information on submap layout algorithms.

The NetView for AIX graphical user interface uses the submap layout algorithm to determine symbol placement.

Controlling Symbol Position

You can override the default symbol placement and programmatically specify a particular position for the symbol. You can specify one of the following position types:

No Position
The symbol has no specific position on the submap. The NetView for AIX program will use the default symbol placement algorithm if you choose this placement value. This position is valid for any layout algorithm.

Coordinate Position
The symbol has an X, Y coordinate position within a grid of given width and height. This position is valid for any layout algorithm.

Sequence Position
The position is specified as part of a sequence, relative to its predecessor. The display of the sequence differs for bus, row/column, star, and ring layouts. A special value is used to indicate the first symbol in the sequence. This position is valid only for ring, star, bus, or row/column layout algorithms.

Star Center Position
The symbol is the center of a star layout. This position is valid only for the star layout algorithm.

Note that not every position form is valid for every submap default layout algorithm. Specifically, the Sequence Position and Star Center Position are valid for only some layout algorithms. You should use care in selecting a position form that is compatible with your submap default layout algorithm.

Guidelines for Symbol Placement

Use the following guidelines when using explicit symbol placement:

- If the submap represents a logical or semantic view of its object, place the symbols according to the rules that apply to the logical or semantic relationship.
- If the submap represents a physical view of its object, place the symbols according to their correspondence to the physical objects.
- If the submap contains multiple semantics and you cannot provide an accurate mapping from symbols to objects, let the NetView for AIX program place the symbols for you.

Setting Sequence Position

Symbols that use any of the sequence layout algorithms (ring, star, bus, row/column) are related to each other by positional order. Each symbol is related to the previous symbol in the sequence by the pred_symbol field in the OVwSymbolPosition data structure. Setting sequence position involves placing a symbol at a particular place in the sequence. To override the NetView for AIX program's automatic symbol placement, specify the symbol ID of the predecessor symbol in the pred_symbol field in the OVwSymbolPosition data structure. This data structure is described in "Creating Icon Symbols" on page 114. The first symbol in the sequence has the special value **ovwNullSymbolId**. The NetView for AIX program will place the symbol in the sequence immediately after the specified predecessor symbol. Depending on whether the user has enabled or disabled automatic symbol layout, the symbol may or may not be placed immediately.

Setting Coordinate Position

Use coordinate placement to position symbols relative to one another or at some fixed point on a background graphic. Submaps without background graphics are scaled so that unused space on the periphery of the submap is not displayed. This does not happen for a submap with a background graphic, because symbol placement is relative to the background graphic and the entire background graphic is always displayed. “Using Submap Background Graphics” on page 142 provides more information on submap background graphics. Specifying symbol position by coordinate position requires two sets of values:

- A width and height to specify a coordinate system, or grid.
- X and Y coordinates on the specified coordinate system where the symbol should be placed.

To define the coordinate system according to which symbols will be placed, you do not need to know the current virtual size of the submap or its screen size when scaled for display. The NetView for AIX graphical user interface translates the grid coordinate position to a virtual position based on the virtual size of the submap. Under certain circumstances, the grid specification affects the virtual size of the submap.

For example, a symbol placed at position (100,100) on a 200x200 grid will be placed at the center of the submap. A symbol placed at position (150,200) on a 300x400 grid will also be placed in the center of the submap. For best results, use the same grid size for all symbols placed on one submap.

You can use the grid size to determine how large symbols appear on the submap. For example, two symbols placed at positions (25,25) and (75,75) on a 100x100 grid will appear in the same positions relative to one another as symbols placed at positions (250,250) and (750,750) on a 1000x1000 grid. Symbols in the latter case will appear smaller because of the greater distance between them. Symbol placement by coordinates takes effect immediately, regardless of whether automatic layout is enabled or disabled.

Note that symbol positions specified by coordinates are lost whenever the user invokes the automatic layout algorithm. If your application needs to disable all automatic layout, it should create the submap with the layout algorithm set to ovwNoLayout.

The following code segment shows how to define coordinate position using the OVwSymbolPosition data structure:

```
...
OVwSymbolPosition *position;

position->placement = ovwCoordPosition;
position->un.coords.x = 50;
position->un.coords.y = 30;
position->un.coords.width = 100;
position->un.coords.height = 90;
...
```

Refer to the OVwSetSymbolPosition() man page if you need more details about these forms of symbol placement.

Creating Bitmaps for NetView for AIX Symbols

NetView for AIX symbols are created by defining symbol classes and subclasses. As described in “Symbol Variety” on page 100, the symbol class defines the shape of the symbol and the subclass defines the bitmap imposed on that shape. This section describes one way to create symbol bitmaps.

Each bitmap is defined by two files: bitmap file and a bitmap mask file. You can use any standard bitmap editing tool to create subclass bitmaps. The following examples show the use of the Bitmap Editor for this purpose.

Symbol Sizes

The NetView for AIX program uses different bitmap sizes based on the dimensions of the NetView for AIX window in which the symbol appears. The NetView for AIX graphical user interface attempts to scale the window contents appropriately, which may mean choosing a different symbol bitmap to optimally fit the window. You should provide a pair of bitmap/bitmap mask files for each size needed by the application. If the appropriate bitmap file does not exist, the NetView for AIX program will choose the closest possible match. It is recommended that you provide bitmaps in the following six sizes (in pixels):

- 20x20
- 26x26
- 32x32
- 38x38
- 44x44
- 50x50

Do your initial design in a medium size, such as 32x32, and then ensure that your symbol will appear properly in the remaining sizes.

Creating Bitmaps

You can view the symbols provided with the NetView for AIX program by selecting **Help..Legend** from the main menu. If there is an existing symbol that is similar to the one you want to create, you can begin by copying that symbol's bitmap and bitmap mask files into files with your name. The NetView for AIX program expects bitmaps to be in the directory **/usr/OV/bitmaps/C**. This is where the bitmaps provided with the NetView for AIX program are located.

The Bitmap Editor can be used to create the bitmap and bitmap mask files. The format of the **bitmap** command is as follows:

```
bitmap [-options ...] filename [WIDTHxHEIGH]
```

More information about the valid options can be obtained by entering the **bitmap** command in your xterm window with no parameters.

The NetView for AIX program uses the following naming convention for naming bitmap and mask files: **name.size.p** (the bitmap) and **name.size.m** (the bitmap mask). To create a new bitmap file called *mb* with a size of *32*, enter:

```
bitmap mb.32.p 32x32
```

Designing Your Symbol

You can think of your symbol as composed of a grid of squares. Each square can be one of three colors:

- Black
- White
- Transparent (the color of the background)

The color of each square is determined by combining the bitmap and bitmap mask files, as described below.

The Bitmap File

When editing the bitmap file, set the squares to black that are to appear black in the symbol. Squares that are to appear either as white or transparent should remain white. The mask file determines whether the white squares end up being displayed as white or transparent.

Figure 3 shows a sample symbol bitmap.

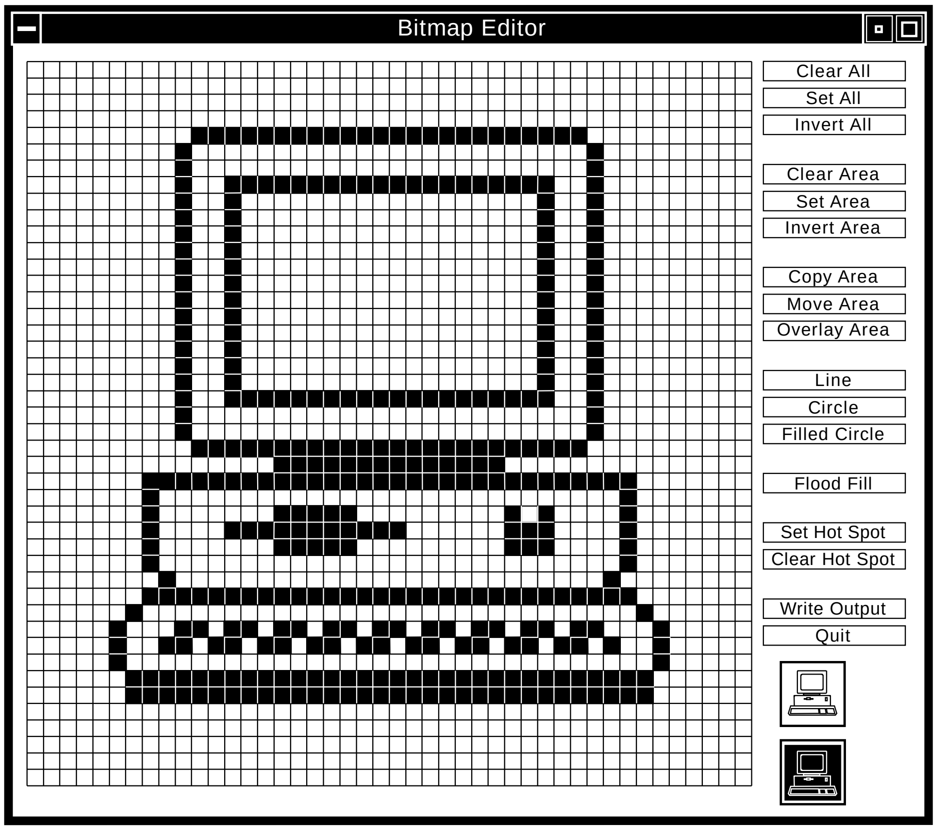

Figure 3. Symbol Bitmap

Bitmap Mask File

When editing the mask file, set the squares to black that you want to appear as black or white in the symbol. Any square that is black in the bitmap mask will show the square as it is in the bitmap file; any square that is white in the bitmap mask will be transparent.

An easy way to get started using bitmap masks is to copy your bitmap file and blacken all the squares within your symbol in the bitmap mask file. This will give you a mask that matches your bitmap file. You can then experiment with changing areas in your bitmap mask file to cause squares to be transparent.

Figure 4 shows a sample symbol bitmap mask.

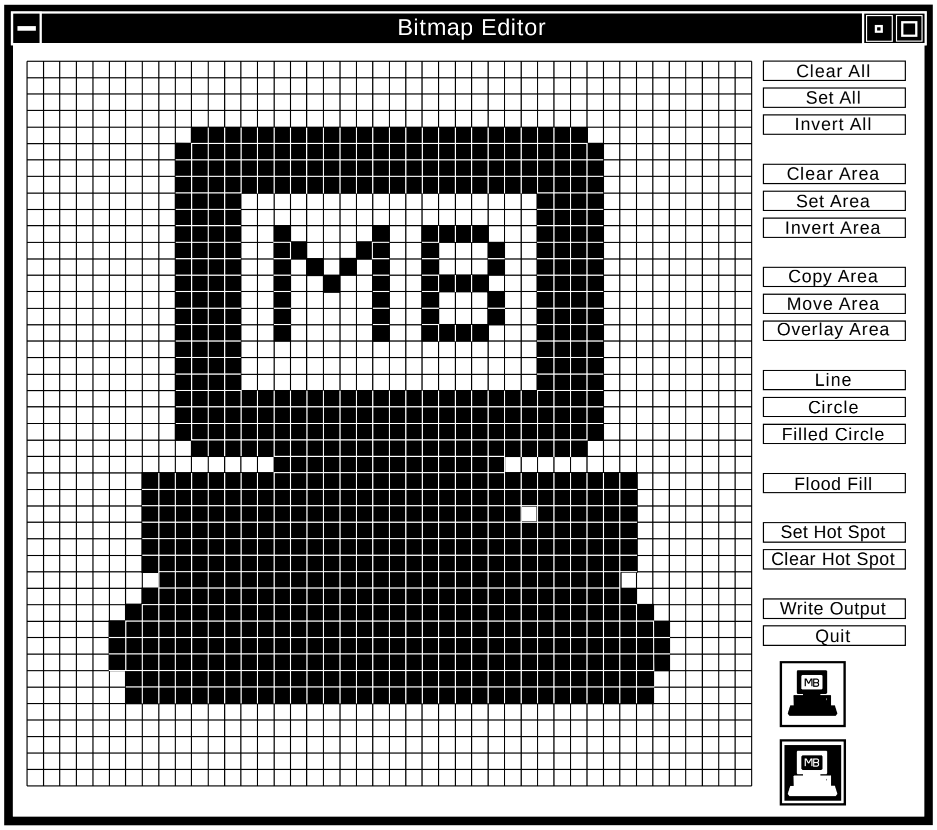

Figure 4. Symbol Bitmap Mask

Combining the Files

The bitmap is created by overlaying the bitmap file with the mask file.

Table 10 on page 108 illustrates the results of the possible combinations of bitmap and mask coding:

Table 10. Bitmap and Bitmap Mask Coding Results

Bitmap	**Mask**	**Result**
Black	Black	Black
Black	White	Transparent
White	Black	White
White	White	Transparent

Figure 5 shows how the NetView for AIX program would display the bitmap shown in the previous examples. The letters MB appear in white because they were black in the mask file and white in the bitmap file. The letters would have appeared in black had they also been black in the bitmap file.

Figure 5 shows the symbol formed by combining the bitmap and the bitmap mask files shown above.

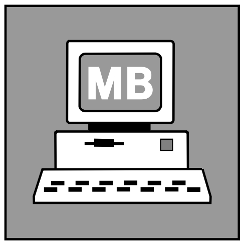

Figure 5. Symbol Created with Bitmap and Bitmap Mask

Bitmap Compilation

Invoke the nv6000 -config command to compile the bitmaps in the /usr/OV/bitmaps/C directory. This creates a file for each bitmap (the name used for the bitmap and mask files without extensions). Using this command enables the bitmaps to be read in quickly during startup of the NetView for AIX program.

Displaying the Bitmap

To display the bitmap, create an NetView for AIX Symbol Type Registration File in the /usr/OV/symbols/C directory as described in "Defining Symbols with Symbol Type Registration Files" on page 109. This file should contain a subclass definition that has a filebase set to the name of the bitmap you created. Then, to see the symbol displayed with the bitmap, bring up the NetView for AIX program and either view the symbol through the Help menu (**Help..Legend**) or add an object (**Edit..Add..Object**).

A simple way to test the creation of the icon without having to start the NetView for AIX program is by using the **xsetroot** command. This lets you change the pointer cursor to an icon when the pointer cursor is outside of any window. This can be done by entering:

```
xsetroot -cursor <bitmap name> <mask name>
```

For example:

```
 xsetroot -cursor mb.44.p mb.44.m
```

Then move the cursor to a position which is not within a window and the cursor shape will change to the new icon.

Defining Symbols with Symbol Type Registration Files

Symbol type registration files are used to define symbol classes and subclasses for the NetView for AIX program. The predefined NetView for AIX symbol classes and subclasses are defined using various symbol type registration files in the **/usr/OV/symbols/$LANG** directory. We encourage you to use these predefined symbol classes and subclasses in your applications.

If the existing classes and subclasses are not adequate for your needs, you can add new symbol subclasses to existing symbol classes, or you can define your own symbol classes. This portion of the chapter shows how to use symbol type registration files to define symbol classes and subclasses. Use symbol type registration files to add symbols that you will use more than once. For temporary symbols, you can use the NetView for AIX symbol routines described later in this chapter. Only icon symbol classes and subclasses are described here, because they are the most commonly used symbols. All references to the terms *symbol classes* and *symbol subclasses* throughout the remainder of this section refer to icon symbol classes and subclasses.

Defining Icon Symbol Classes

Symbol classes are defined using files in the **/usr/OV/symbols/$LANG** directory. Each file in the symbols directory contains one or more symbol class definitions, where each definition is composed of three elements:

- The class name (a string)
- The description of the external shape (a series of line segments or arcs)
- An optional integer scale factor

Complete examples of symbol class definitions are provided in the following sections. Symbol class definitions based on line segments are treated first, followed by class definitions based on arcs.

Defining Classes Using Line Segments

The majority of icon symbol classes provided with the NetView for AIX program have straight sides, indicating that they have been defined using line segments. To define a symbol class using line segments, imagine the shape imposed on a graph. You define the shape by specifying the coordinates of the end-points of the line segments. All points must be defined with integer values. For example, to define a simple square (the Computer symbol class), specify the following definition:

```
SymbolClass "Computer {
   Segment (-1,1) to (1,1) to (1,-1)
           to (-1,-1) to (-1,1);
}
```

The point (0,0) must lie within the outline of your symbol; otherwise the NetView for AIX program will not be able to make correct connections between this symbol and others on the submap.

By default, the scale is from -1 to 1 on both the horizontal and vertical axes. This scale limits the shapes you can define. To make more complex shapes, you can increase the scale. The following example illustrates the use of the Scale parameter:

```
SymbolClass "House" {
   Scale 5;
   Segment (-4,0) to (-5,0) to (0,5) to
      (3,2) to (3,3) to
      (4,3) to (4,1) to (5,0) to
      (4,0) to (4,-5) to (-4,-5);
}
```

This code produces the icon shape shown in Figure 6.

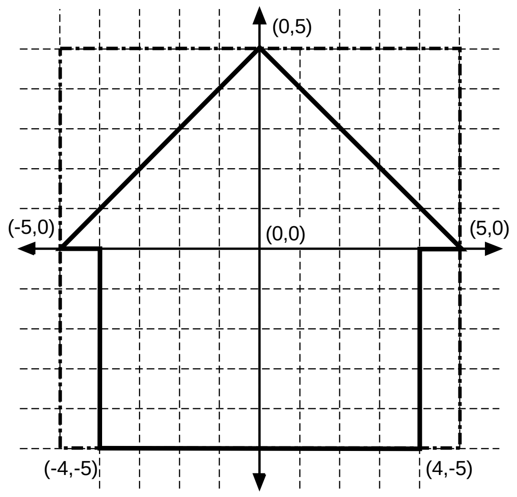

Figure 6. House Symbol Class

Specifying a higher scale value enables you to draw more complex shapes. The NetView for AIX program will interpret the points you specify according to the scale. To produce a slightly larger symbol, specify points that exceed the specified scale value. For example, to make the house in the previous example taller, you could specify the two bottom corners as (4,-7) and (-4,-7). If you use this technique, be sure not to exceed 150% of your specified scale.

Note that the example segment definition is not explicitly closed. The NetView for AIX program completes the definition by drawing a line from the last point to the starting point.

Defining Classes Using Arcs

As an alternative to using line segments, you may also define icon symbol classes using arcs. Arcs allow you to define class shapes such as circles and ellipses that would be tedious to define using line segments. The definition of an arc is composed of three components:

Size This is the width and height of a conceptual rectangle in which the line will rotate. If the width and height are equal, the arc will have a constant radius. Elliptical arcs may be achieved using height and width values that are unequal.

Origin Specifies an origin for rotation within the conceptual rectangle. The center of the conceptual rectangle is assumed to have the coordinates (0,0).

Rotation This specifies the starting angle and number of degrees of rotation. A starting angle of 0 degrees is equated to a three-o'clock position, and rotation occurs in a clockwise direction.

The following example shows how to define a symbol class with a circular shape:

```
SymbolClass "Network" {
    Scale 2;
    Arc Origin(0,0) Size(2,2) Rotation 0, 360;
}
```

The following example demonstrates how to define a symbol class shaped like an ellipse that is wider than it is high:

```
SymbolClass "Network" {
    Scale 4;
    Arc Origin(0,0) Size(4,2) Rotation 0, 360;
}
```

Note: Arcs and segments cannot be combined in a single shape definition.

Optional Class Specifications

There are four optional class specifications you can use when you define symbol classes.

- Default Layout

  When a user double-clicks on an executable symbol on a map, the symbol explodes into a submap. Use the **DefaultLayout** entry to specify the default layout algorithm for the child submap of an object represented by a symbol with this symbol class. This default can be overridden by a user or an application when the submap is created. The choices are: **Ring, Bus, Star, PointToPoint,** and **RowColumn** (the default).

- Default Status Source

  A symbol can get status from any of the following sources:

  – From the underlying object (object status)

  – From the parent object of a child submap (compound status)

  – From the symbol itself (symbol status)

  You can assign default status source to symbol classes by defining a **DefaultStatusSource** entry in the class definition. Supply one of three status source values: **Object, Symbol,** or **Compound**. The following example shows how to set the default layout and the default status source for a symbol class:

  ```
  SymbolClass "Computer" {
      Segment (-1,-1) to (1,-1) to (1,1) to (-1,1);
      DefaultLayout RowColumn;
      DefaultStatusSource Compound;
  }
  ```

  If this entry is not set, symbols in this class will derive their status from the objects they represent. See “Symbol Status” on page 101 for more details about symbol status.

- Default Capabilities

  An object can be added to the map by selecting a symbol type on the symbol palette and dragging it to a submap. The NetView for AIX program automatically creates an object that is associated with the symbol. By default, the new object has no capabilities. Capabilities are described in “Field Flags” on page 81.

You can assign default capabilities to objects by defining a **Capabilities** statement in your class definition. Whenever a symbol is added to a map, the NetView for AIX program will use the symbol's default capabilities to initialize the capabilities of the associated object being added as shown in the following example:

```
SymbolClass "Computer"
{
   Segment (-1,-1) to (1,-1) to (1,1) to
   (-1, 1) to (-1,-1);
   Capabilities {
   isNode = 1;
   }
}
```

- Variety

  There are two forms of symbol classes: icons and connections. To this point, only icons have been shown. It is possible, though, to create symbol classes that connect other symbols. These are called connection classes. You can specify that a class is a connection by using the **Variety** definition and specifying that the class has the value **Connection**. (The default is **Icon**). The following example demonstrates:

```
SymbolClass "Lines" {
    Variety Connection;
}
```

  For further information about connection symbol classes, see the OVwRegIntro() man page.

Defining Symbol Subclasses

A Symbol Subclass definition enables you to define a new symbol subclass within a class of symbols. Subclasses are represented using bitmaps that are superimposed on symbol class shapes. You can use any standard bitmap editing tool to create subclass bitmaps.

Symbol subclasses are defined using a SymbolType block. The SymbolType block specifies the name of the subclass and the class to which it belongs. The symbol type can be considered the combination of the symbol class and the symbol subclass. It has the following syntax:

```
SymbolType "class name": "subclass name" {
    ...
    subclass specifications
    ...
}
```

The `class name` is the name of an existing symbol class. The `subclass name` is a unique name for the new subclass. The subclass specifications define the bitmaps that are used for the subclass as well as other characteristics of symbol subclass behavior. Some subclass specifications are required, and others are optional.

Required Subclass Specifications

Two subclass specifications must be present in every icon symbol subclass definition:

Filebase This entry defines the base name for a symbol subclass bitmap. A bitmap definition is composed of two parts: **filebase.size.p** (the bitmap) and **filebase.size.m** (the bitmap mask). You should provide a pair of bitmap/bitmap mask files for each bitmap size your application supports. You must provide at least one bitmap pair.

The NetView for AIX program uses different bitmap sizes whenever the dimensions of an NetView for AIX window change. The NetView for AIX graphical user interface attempts to scale the window contents appropriately, which may mean choosing a different symbol bitmap to optimally fit the window. If the appropriate bitmap file does not exist, the NetView for AIX program will choose the closest match. Provide bitmaps in the following sizes (in pixels): `20x20`, `26x26`, `32x32`, `38x38`, `44x44`, and `50x50`.

Cursor Size Whenever a user selects a symbol type from the symbol palette and moves the symbol type to a map, the cursor shape changes from a pointer to a bitmap. Use the CursorSize entry to define the size of the bitmap to be used as the cursor. Use a cursor size of 38x38 pixels.

The following example illustrates both the Filebase and CursorSize entries:

```
SymbolType "Software": "Application" {
    Filebase "app2";
    CursorSize 38;
}
```

In this example, the cursor will be 38x38 pixels and will be based on the bitmap formed by the files **app2.38.p** and **app2.38.m**.

Optional Subclass Specifications

In addition to the required subclass specification, you can use optional subclass specifications to further define your symbol subclass. Use these optional specifications to override the values that you specified when defining the symbol class. The following values may be specified:

- Default layout
- Default status source
- Default capabilities

If these specifications are not coded in the subclass definition, the values from the class definition are inherited by the subclass. If these specifications are coded in the subclass definition, they override, for that subclass only, the values specified in the class definition.

For further symbol subclass examples, see the registration files provided with the NetView for AIX program, in the /usr/OV/symbols/$LANG directory.

Defining Symbols with NetView for AIX EUI Routines

Creating symbols with Symbol Type Registration Files gives you stability and simplifies your application coding. However, there may be times when you need the flexibility of being able to create, modify, or delete symbols while your application is running. The NetView for AIX EUI API provides several routines that enable you to work with symbols from within your application.

Creating Icon Symbols

The OVwCreateSymbol() routine is a general purpose routine that can create any type of icon symbol for an existing object. It has the following function prototype:

```
OVwSymbolId OVwCreateSymbol( OVwMapInfo *mapInfo, OVwSubmapId submapId,
        OVwObjectId objectId, OVwSymbolType symbolType, char *label,
        OVwStatusType status, int statusSource,
        OVwSymbolPosition *symbolPosition, unsigned int flags )
```

When you call the OVwCreateSymbol() routine, you must provide the following arguments:

mapInfo	Points to an OVwMapInfo data structure that contains complete map information. Most NetView for AIX API symbol routines take a pointer to the mapInfo data structure as the first argument. You can use the OVwGetMapInfo() routine to generate a mapInfo data structure, or you can save the map information in a callback routine that handles the map open event. Within that callback routine, you can use the OVwCopyMapInfo() routine to save the map information returned with the ovwMapOpen event. Map information is valid until the map is closed.
submapId	Identifies which submap the symbol should be placed on. Submaps are described in Chapter 9, “Creating and Using Submaps” on page 131.
objectId	The object ID of the associated underlying object.
symbolType	A character string identifying a symbol class and subclass as defined in a symbol type registration file. The header file <OV/sym_types.h>, which is automatically included by the <OV/ovw.h> header file, contains localized string definitions for symbol types shipped with the NetView for AIX program.
label	A character string that is initially displayed at the bottom of the symbol. Users can change the label at any time, and both users and developers can optionally disable the display of the label.
status	The initial status of the symbol. Possible values are: • ovwUnknownStatus • ovwNormalStatus • ovwMarginalStatus • ovwCriticalStatus • ovwUnmanagedStatus
statusSource	Defines how the symbol receives status information. Possible values are: • ovwSymbolStatusSource

- ovwObjectStatusSource
- ovwCompoundStatusSource

symbolPosition Specifies where the symbol is to be placed on the submap. A NULL value indicates that the symbol placement algorithm for the submap should be used. Although the default position is adequate in most cases, you can specify a particular position for a symbol you create. You can specify any of the following values:

- ovwNoPosition
- ovwCoordPosition
- ovwSequencePosition
- ovwStarCenterPosition

See "Controlling Symbol Position" on page 103 for descriptions of these symbol placements and the layout algorithms with which each can be used.

You can specify these position forms, as well as the associated information for the position, using the following data structure:

```
typedef struct {
   int placement;
   union {
      struct {
         int x;
         int y;
         int width;
         int height;
      } coords;
      OVwSymbolId pred_symbol;
   } un;
} OVwSymbolPosition;
```

The placement field is set to one of the four position forms. If you use either the **ovwCoordPosition** or **ovwSequencePosition**, you will also need to set the appropriate values within the union.

Note: If automatic layout is disabled, the symbol may not be placed until automatic layout is reenabled. See the OVwSetSymbolPosition(3) man page if you need precise information about how the NetView for AIX program treats explicit symbol placement for the various layout algorithms when automatic layout is enabled or disabled.

flags Specifies flags which control several aspects of symbol creation. You can use one or more of the following flags when you create a symbol:

ovwNoSymbolFlags
Setting this flag is equivalent to passing a NULL value for the flags parameter to OVwCreateSymbol().

ovwDoNotDisplayLabel
Setting this flag will prevent the symbol label from being displayed. This flag is typically used when creating connection symbols.

ovwMergeDefaultCapabilities
Setting this flag will merge the default capability fields of the symbol type specified in the OVwCreateSymbol() call with the set of capability fields for the object. If a capability field exists in the symbol type definition, but not in the object definition, it will be added to the object. Existing field values within the object will not be changed.

ovwDeleteDescendants
When this symbol is deleted, all other symbols that represent this same object on submaps descending from this symbol's submap will also be deleted. This flag is useful for applications that build a submap hierarchy with symbols representing the same object appearing on several submaps. This flag facilitates deletion of the object, since all symbols representing an object must be deleted before the object can be deleted.

See the OVwCreateSymbol() man page if you need more information about these flags.

The OVwCreateSymbol() routine returns a symbol ID, which is a unique numeric identifier for the symbol. The symbol ID is used in subsequent calls involving the symbol.

The following code segment shows how to create a symbol. The ID of the object that the symbol represents is passed as an argument.

```
create_symbol( object_id )
OVwObjectId object_id;
{
   OVwSymbolId symid;
   OVwMapInfo *map;

   map = OVwGetMapInfo();

   /* Create the symbol, using an NetView for AIX
   constant to represent the symbol type */
   symid = OVwCreateSymbol( map, map->root_submap_id, object_id,
           ovwSWorkstationComputer, "test", ovwUnknownStatus,
           ovwSymbolStatusSource, NULL, ovwNoSymbolFlags );
   ...
   OVwFreeMapInfo( map );
}
```

In this example, the symbol type for the symbol class/subclass Computer: Workstation is placed on the root submap of the open map. The symbol has the label “test.” The initial status is unknown, and the status is set on a symbol basis.

Creating Symbols with Convenience Routines

Although the OVwCreateSymbol() routine can create any type of icon symbol, convenience routines are available that are easier to use in certain situations. They include:

- OVwCreateSymbolByName()
- OVwCreateSymbolBySelectionName()
- OVwCreateSymbolByHostname()

These routines let you identify the object by one of its names, instead of by its object ID. If the object does not yet exist, these routines will first create the object for you using the OVwDbCreateObject() routine.

- OVwCreateComponentSymbol(),
- OVwCreateComponentSymbolByName()

These two routines let you create symbols on a submap by identifying the parent object of the submap, rather than by identifying the submap ID. If the submap does not exist, the NetView for AIX program will create it for you. These convenience routines use many of the same parameters as the general purpose OVwCreateSymbol() routine and are quite similar in concept. See the man pages if you need more information about these routines.

Creating Connection Symbols

Creating a connection symbol is very similar to creating an icon symbol. Many of the parameters are similar to those used to create an icon symbol. Use the OVwCreateConnSymbol() routine to create a connection symbol. This routine has the function prototype:

```
OVwSymbolId OVwCreateConnSymbol( OVwMapInfo *mapInfo,
              OVwObjectId objectId,
              OVwSymbolId endpoint1, OVwSymbolId endpoint2,
              OVwSymbolType symbolType,
              char *label, OVwStatusType status,
              int statusSource, unsigned int flags );
```

An object that represents the connection must already exist in order to use the OVwCreateConnSymbol() routine. The *objectId* argument contains the object ID of the object that represents the connection. The *endpoint1* and *endpoint2* parameters specify the symbol IDs of the icon symbols to be connected. The special value **ovwSubmapBackbone** may be substituted for one of the end points if the submap layout algorithm uses a backbone. The NetView for AIX program supports two layout algorithms that use backbones: **ovwBusLayout** and **ovwRingLayout**.

The symbolType parameter must specify a symbol type belonging to the Connection symbol class. For convenience, you can use a value of NULL for symbolType to indicate the default symbol type. (A NULL symbol-type value is not permitted when creating icon symbols.)

Connection Symbols and Metaconnections

When the first connection is created between two symbols, a simple connection symbol is added to the submap. When a second connection is made between the two symbols, a special metaconnection submap is automatically created by the NetView for AIX program to hold the two connections and a metaconnection symbol replaces the original simple connection. Double-clicking on the metaconnection symbol displays the metaconnection submap showing all connections between the two symbols.

Application developers do not need to be concerned with metaconnection symbols when creating connections. NetView for AIX manages metaconnections for you automatically.

Developers must be prepared to deal with metaconnections, though, when reading symbol information. The metaconnection, not the actual connection, appears in references to connection symbols in the submap containing the connection. For example, assume that you need to modify the status of a connection symbol. When you get a list of all symbols on a submap, you would find a metaconnection symbol, not the actual connection symbol, for the particular connection. You would need to get the submap associated with the metaconnection symbol and check that submap for the presence of your connection symbol. After finding your connection symbol in the metaconnection submap, you could change the status of the connection symbol.

Refer to the OVwCreateSymbol() man page if you need more information about metaconnections.

Creating Several Symbols with a Single Call

The NetView for AIX program provides the OVwCreateSymbols() routine to let you create several symbols in a single call. The OVwCreateSymbols() routine can create combinations of icon and connection symbols. To do this, the OVwCreateSymbols() routine uses a rather complex data structure that enables you to define either icon or connection symbols, or both. None of the data structure elements are new; they have all been described in some form earlier in this chapter. The OVwCreateSymbols() routine has the function prototype:

```
int OVwCreateSymbols( OVwMapInfo *mapInfo,
                      OVwSymbolCreateList *symbolList );
```

The *symbolList* parameter points to a data structure of type OVwSymbolCreateList that contains a count of the number of symbols and a pointer to an array of OVwSymbolCreateInfo data structures, each of which fully defines the symbol to be created. The OVwSymbolCreateList data structure is shown in the following example:

```
typedef struct {
   int count;                    /* number of items in the list */
   OVwSymbolCreateInfo *symbols;  /* contiguous list of items */
} OVwSymbolCreateList;
```

Use the OVwSymbolCreateInfo data structure to define each symbol you wish to create, as shown in the next example:

```
typedef struct {
   int submap_name_style;
   union {
      OVwSubmapId submap_id;
      OVwObjectId parent_id;
   } submap_un;
   int object_name_style;
   union {
      OVwObjectId object_id;
      OVwFieldBinding *object_name;
   } obj_un;
   char *label;
   OVwStatusType status;
   int status_source;
   OVwSymbolType symbol_type;
   int symbol_variety;
   union {
      OVwSymbolPosition *position;
      struct {
         int endpoint1_name_style;
         union {
            OVwSymbolId symbol_id;
            int symbol_index;
         } un_endpoint1;
         int endpoint2_name_style;
         union {
            OVwSymbolId symbol_id;
            int symbol_index;
         } un_endpoint2;
      } endpoints;
   } sc_var_un;
   unsigned int flags;
   char *object_comments;
   int error;
   OVwSymbolId symbol_id;
} OVwSymbolCreateInfo;
```

The following example shows how to create multiple symbols with the OVwCreateSymbols() routine:

```
#include <OV/ovw.h>

#define BUILD_ICON(sym,sid,oid,stype,slabel,stat,src,pos,flgs) { \
       sym.submap_name_style = ovwSubmapIdValue; \
       sym.sc_submap_id = sid; \
       sym.object_name_style = ovwObjectIdValue; \
       sym.sc_object_id = oid; \
       sym.label = slabel; \
       sym.status = stat; \
       sym.status_source = src; \
       sym.symbol_type = stype; \
       sym.symbol_variety = ovwIconSymbol; \
       sym.flags = flgs; \
       sym.sc_icon_position = pos; \
       sym.object_comments = "No comment"; \
       sym.error = 0; \
       sym.symbol_id = ovwNullSymbolId; \
}
```

```
#define BUILD_CONNECTION(sym, oid, stype, slabel, svalue,src,sym1,sym2,flgs) {\
       sym.object_name_style = ovwObjectIdValue; \
       sym.sc_object_id = oid; \
       sym.label = slabel; \
       sym.status = svalue; \
       sym.status_source = src; \
       sym.symbol_type = stype; \
       sym.symbol_variety = ovwConnSymbol; \
       sym.flags = flgs; \
       sym.sc_conn_endpoint1_style = ovwEndpointSymbolIdValue; \
       sym.sc_conn_endpoint1_id = sym1; \
       sym.sc_conn_endpoint2_style = ovwEndpointSymbolIdValue; \
       sym.sc_conn_endpoint2_id = sym2; \
       sym.object_comments = NULL; \
       sym.error = 0; \
       sym.symbol_id = ovwNullSymbolId; \
}


void
CreateSymbol(void *, char *actionID, char *menuItemID,
OVwObjectIdList *, int , char **)
{
   OVwSymbolId symbol_id;
   OVwObjectId object_id;
   OVwSymbolInfo *symbol;
   OVwSymbolCreateList symbolList;
   OVwSymbolCreateInfo symbolInfo;
   OVwMapInfo *map = OVwGetMapInfo();

   fprintf(stderr, "ACTION: %s\n", actionID);
   fprintf(stderr, "MENUITEM: %s\n", menuItemID);

   symbolList.count = 1;
   symbolList.symbols = &symbolInfo;

   object_id =  OVwDbCreateObject(NULL);
   BUILD_ICON(symbolInfo, map->root_submap_id, object_id,
              "Computer:Workstation", "new symbol", ovwNormalStatus
              ovwSymbolStatusSource, NULL, ovwMergeDefaultCapabilities);

   OVwCreateSymbols(map, &symbolList);

   symbol_id = symbolList.symbols[0].symbol_id;
   symbol = OVwGetSymbolInfo(map, symbol_id);
   OVwPrintSymbolInfo(symbol, TRUE);

    OVwFreeMapInfo(map);
}


void
CreateConnection(void *, char *actionID, char *menuItemID,

OVwObjectIdList *, int, char **)
{
   OVwSymbolCreateList symbolList;
   OVwSymbolCreateInfo symbolInfo[3];
   OVwObjectId object_id;
   OVwSymbolId symbol_id;
   OVwSymbolId symbol1_id;
   OVwSymbolId symbol2_id;
   OVwSymbolInfo *symbol;
   OVwMapInfo *map = OVwGetMapInfo();
   fprintf(stderr, "ACTION: %s\n", actionID);
   fprintf(stderr, "MENUITEM: %s\n", menuItemID);

   symbolList.count = 2;
   symbolList.symbols = &symbolInfo[0];

   object_id = OVwDbCreateObject(NULL);

   BUILD_ICON(symbolInfo[0], map->root_submap_id,
      object_id, "Computer:Workstation",
      "number one", ovwMarginalStatus,
      ovwObjectStatusSource, NULL, 0);

   BUILD_ICON(symbolInfo[1], map->root_submap_id,
      object_id,"Computer:Workstation",
      "number two", ovwMarginalStatus,
      ovwObjectStatusSource, NULL, 0);

   OVwCreateSymbols(map, &symbolList);

   symbol1_id = symbolList.symbols[0].symbol_id;
   symbol = OVwGetSymbolInfo(map, symbol1_id);
   OVwPrintSymbolInfo(symbol, TRUE);

   symbol2_id = symbolList.symbols[1].symbol_id;
   symbol = OVwGetSymbolInfo(map, symbol2_id);
   OVwPrintSymbolInfo(symbol, TRUE);

   symbolList.count = 1;

   BUILD_CONNECTION(symbolInfo[0], object_id, NULL, NULL,
                  ovwNormalStatus, ovwObjectStatusSource,
                  symbol1_id, symbol2_id, 0);

   OVwCreateSymbols(map, &symbolList);

   symbol_id = symbolList.symbols[0].symbol_id;
   symbol = OVwGetSymbolInfo(map, symbol_id);
   OVwPrintSymbolInfo(symbol, TRUE);

   symbol = OVwGetConnSymbol(map, symbol1_id, symbol2_id);
   OVwPrintSymbolInfo(symbol, TRUE);

    OVwFreeMapInfo(map);
}


main(int, char **)
{
   if (OVwInit() << 0) {
       fprintf(stderr, "%s\n", OVwErrorMsg(OVwError()));
        exit(1);
    }

   OVwAddActionCallback("Symbol",
                    (OVwActionCallbackProc)CreateSymbol, NULL);
   OVwAddActionCallback("Connection",
                    (OVwActionCallbackProc)CreateConnection, NULL);

    OVwMainLoop();
}
```

The advantage of using this routine is that it takes less execution time to create several symbols with OVwCreateSymbols() than it does to create them individually. The disadvantages of using OVwCreateSymbols() are that the data structures are more complex, and users can encounter delays while long lists of symbols are being created.

In general, if users need timely, continuous feedback, consider using the OVwCreateSymbol() routine or use OVwCreateSymbols() to create only a few symbols at a time. If users do not expect immediate feedback, you can use the more efficient OVwCreateSymbols() routine.

Changing Symbol Appearance and Behavior

Once a symbol exists, you can manipulate it in the following ways:

- Change the symbol type (the icon or connection graphic).
- Override NetView for AIX's symbol placement algorithms and manually place the symbol yourself.
- Change the symbol behavior to be explodable or executable.
- Change the symbol label.
- Change the status source for the symbol.
- Set or clear application interest in a symbol.

Style considerations may apply to these operations. Refer to the *NetView for AIX Application Interface Style Guide* for complete guidelines on symbol manipulation.

Changing a Symbol's Type

You can change a symbol's type at any time. Use the OVwSetSymbolType() routine to change the symbol type. It has the following function prototype:

```
int OVwSetSymbolType( OVwMapInfo *mapInfo, OVwSymbolId symbolId,
               OVwSymbolType symbolType, unsigned int flags);
```

You can use application-defined symbol types, or you can use the symbol types provided with the NetView for AIX program. The header file <OV/sym_types.h>, which is automatically included by the <OV/ovw.h> header file, contains localized string definitions for symbol types shipped with the NetView for AIX program.

Changing a Symbol's Position

You can change the position of a symbol at any time using the OVwSetSymbolPosition() routine. This routine changes the position of an existing symbol. It has the following function prototype:

```
int OVwSetSymbolPosition( OVwMapInfo *mapInfo, OVwSymbolId symbolId,
               OVwSymbolPosition *position );
```

The OVwSymbolPosition data structure is described in “Creating Icon Symbols” on page 114. That section tells how to use the OVwSymbolPosition data structure to define symbol position.

Changing a Symbol's Behavior

Use the OVwSetSymbolBehavior() routine to change the behavior of a symbol on the open map. By default, symbols are explodable. This routine can make a symbol explodable or executable. The OVwSetSymbolBehavior() routine has the function prototype:

```
int OVwSetSymbolBehavior( OVwMapInfo *mapInfo,
            OVwSymbolId symbolId, int behavior, char *appName,
            char *actionId, OVwObjectList *targetObjects );
```

The OVwSetSymbolBehavior() routine has the following arguments:

mapInfo	Points to an OVwMapInfo data structure that contains complete map information. You can use the OVwGetMapInfo() routine to get the map information.
symbolId	The symbol ID of the symbol whose behavior is being changed.
behavior	Set to **ovwSymbolExplodable** or **ovwSymbolExecutable**.
appName	The name of an application registered in an Application Registration File.
actionId	The action to be performed by application appName when the symbol is executed. The *actionId* argument must name a valid action registered in the application registration file. Actions are described in “Defining Application Invocation” on page 36
targetObjects	An optional list of object IDs to be used as the selection list when the action is performed.

Regardless of whether a symbol is explodable or executable, many of the symbol characteristics always apply. A symbol always has a status source, a status value, a symbol position, an association to an object, and an optional symbol label. Changing symbol behavior does not affect these characteristics.

For example, an explodable symbol that receives status by object will continue to receive status by object, even when changed to an executable symbol. Also, any associations between an underlying object and child submaps remain in effect when an explodable symbol is changed to an executable symbol. Users, however, can no longer navigate to the child submap through the executable symbol.

Note: Use care when changing an explodable symbol to an executable symbol. If the explodable symbol has a child submap attached, making the symbol executable might make it difficult for users to access the child submap in the future. If the child submap is accessible only through a single explodable symbol and that symbol is made executable, users must resort to using the NetView for AIX graphical user interface submap list box to navigate to the submap.

The following code segment demonstrates how to make a symbol executable. Assume that the action ID `Show Users` is already defined in the application registration file for the application “User Admin”:

```
#include <OV/ovw.h>

make_executable( symId )
OVwSymbolId symId;
{
   int ret;
   char *app_name;
   char *action_name = "Show Users";
   OVwMapInfo *map;

   map = OVwGetMapInfo();
   app_name = OVwGetAppName();
   ret = OVwSetSymbolBehavior( map, symId, ovwSymbolExecutable,
                               app_name, action_name, NULL );
   free( app_name );
   OVwFreeMapInfo( map );
   ...
}
```

Note that the OVwGetAppName() routine retrieves the application name before calling OVwSetSymbolBehavior().

Changing a Symbol's Label

Use the OVwSetSymbolLabel() routine to change a symbol's label. The routine has the function prototype:

```
int OVwSetSymbolLabel( OVwMapInfo *mapInfo,
                       OVwSymbolId symbolId, char *label );
```

The OVwSetSymbolLabel() routine is only used to define the label. Use the OVwSetSymbolType() routine to control whether the label is displayed.

Note: You should use discretion when using this routine. In most cases, an application should not change a label that has been modified by the user. It is usually safe to change the label if the application originally set the label and the user has not modified it (the label still has the value set by the application). The OVwGetSymbolInfo() routine, which is described later, can retrieve the symbol label for a given symbol.

Changing a Symbol's Status

You can change a symbol's status in the following ways:

- On the symbol if the status source is **ovwSymbolStatusSource.**
- On the object if the status source is **ovwObjectStatusSource.**

You cannot directly set the status for a symbol that has compound status source. If the symbol has compound status, you can only set symbol or object status on the contained symbols or objects. As a result, the compound status may change. The algorithm for determining how to propagate compound status is managed by the NetView for AIX program.

The NetView for AIX routines that set status values on a symbol or object have the following function prototypes:

```
int OVwSetStatusOnSymbol( OVwMapInfo *mapInfo, OVwSymbolId symbolId,
                          OVwStatusType status );
int OVwSetStatusOnObject( OVwMapInfo *mapInfo, OVwObjectId objectId,
                          OVwStatusType status );
```

OVwSetStatusOnSymbol() changes the status on the specified symbol only if the symbol has status source **ovwSymbolStatusSource**.

OVwSetStatusOnObject() changes the status of an object in the open map. This call also sets the object status on all symbols that represent the object and have status source **ovwObjectStatusSource**. Neither symbol nor object status can be set on an object that is unmanaged.

Two list forms of status routines are also available for setting status on several symbols or objects. They are more efficient than making multiple calls, because the NetView for AIX program's compound status propagation is disabled until the status values of all entities in the list are set. They have the function prototypes:

```
int OVwSetStatusOnSymbols( OVwMapInfo *mapInfo,
                           OVwSymbolStatusList *symbolList );
int OVwSetStatusOnObjects( OVwMapInfo *mapInfo,
                           OVwObjectStatusList *objectList );
```

The *symbolList* and *objectList* data structures have the same form as other common NetView for AIX list structures. They contain a count of the number of elements in the list, and a pointer to the first in a contiguous array of list entries.

Changing a Symbol's Status Source

In addition to changing the status of a symbol or object, you can also change a symbol's status source. Status source is defined when the symbol is created, and is changed at a later time using the OVwSetSymbolStatusSource() routine. It has the following function prototype:

```
int OVwSetSymbolStatusSource( OVwMapInfo, *mapInfo,
                              OVwSymbolId symboldId, int statusSource );
```

The *statusSource* field can contain any of the three status sources described earlier in this manual: **ovwSymbolStatusSource, ovwObjectStatusSource,** or **ovwCompoundStatusSource**.

You should only change status source for symbols that you create or for symbols that users add to submaps that you create. Do not change status source for symbols created by other applications. Further, if end-users have modified the status source for an application created-symbol, you should not alter the user's preference.

Setting or Clearing Application Interest in a Symbol

NetView for AIX maintains a list of all applications interested in each symbol. Any time an application creates a symbol, the list of interested applications is initially set to contain the name of the application. Other applications can add their name to the list of applications interested in that symbol. There are a couple of reasons for applications to express interest in a symbol.

- Symbols can take on a different appearance depending on whether an application has shown interest in the symbol. If one or more applications are interested in a symbol, the symbol appears in the application plane of the submap. If no application is interested, the symbol appears in the user plane.
- Applications can search a submap for all symbols that possess certain criteria. (These routines will be described later in this chapter.) In this case, the search can be filtered to only consider those symbols in which the application has expressed interest.

An application interested in a symbol created by another application should call the OVwSetSymbolApp() routine. This routine adds the calling application to the list of applications interested in the symbol. The OVwClearSymbolApps() routine clears the application from this list. These routines have the function prototypes:

```
int OVwSetSymbolApp( OVwMapInfo *mapInfo, OVwSymbolId symbolId );
int OVwClearSymbolApp( OVwMapInfo *mapInfo, OVwSymbolId symbolId );
```

See the OVwSetSymbolApp() man page if you need more information on these calls.

Retrieving Symbol Information

This section describes the routines that retrieve symbol information. Before describing these routines, it is necessary to describe the OVwSymbolInfo data structure. This data structure is used by most NetView for AIX routines that retrieve symbol information. It supports both icon and connection symbols. Many of the elements within the **OVwSymbolInfo** data structure have already been presented earlier in this chapter.

The OVwSymbolInfo data structure has the following definition:

```
typedef struct {
   OVwSymbolId symbol_id;          /* symbol ID */
   OVwSubmapId submap_id;          /* submap symbol is on */
   OVwObjectInfo object;           /* object represented by the symbol */
   char *symbol_label;             /* symbol label */
   OVwStatusType symbol_status;    /* symbol status */
   int status_source;              /* status source */
   OVwPlaneType plane;             /* submap plane the symbol is on */
   OVwBoolean hidden;              /* whether symbol is hidden on submap */
   OVwSymbolType symbol_type;      /* symbol type */
   int symbol_variety;             /* ovwIconSymbol, ovwConnSymbol */
   union {
      OVwSymbolPosition position;  /* ovwIconSymbol variety  */
      struct {                     /* ovwConnSymbol variety  */
         OVwSymbolId endpoint1_id; /* connection endpoint   */
         OVwSymbolId endpoint2_id; /* connection endpoint */
         OVwBoolean is_meta_conn;  /* metaconnections? */
      } conninfo;
   } var_un;
   int behavior;                   /* ovwSymbolExplodable or */
                                   /* ovwSymbolExecutable */
   OVwApplist apps;                /* apps interested in symbol */
} OVwSymbolInfo;
```

Most of the fields in this data structure have been used in previously-described symbol routines; two fields in particular deserve comment.

- The **symbol_variety** field serves as a tag for the union that follows it (var_un). If the symbol is an icon, the position field contains icon-specific symbol information (the symbol position in this case). If the symbol is a connection, then the **conninfo** structure defines connection-specific symbol information.
- The **is_meta_conn** field in the **conninfo** union indicates whether a connection symbol is a simple connection or a metaconnection. A metaconnection is a special connection that represents multiple connections between the two connected symbols. If the symbol is a metaconnection, the contained simple connections are found on the child submap of the object represented by the metaconnection.

Using the OVwGetSymbolInfo() Routine

The OVwGetSymbolInfo() routine is a general-purpose routine that can retrieve complete symbol information for any symbol. Given a symbol ID for any symbol, this routine returns a pointer to an OVwSymbolInfo data structure containing complete symbol information. An associated OVwFreeSymbolInfo() routine frees memory allocated by the OVwGetSymbolInfo() routine. They have the function prototypes:

```
OVwSymbolInfo *OVwGetSymbolInfo( OVwMapInfo *mapInfo,
               OVwSymbolId symbolId );
void OVwFreeSymbolInfo( OVwSymbolInfo *symbol );
```

The following example shows how to use these routines:

```
#include <OV/ovw.h>

dump_symbol_info( symbol_id )
OVwSymbolId symbol_id;
{
   OVwSymbolInfo *p;
   OVwMapInfo *map;

   map = OVwGetMapInfo();
   p = OVwGetSymbolInfo( map, symbol_id );
   if ( p == NULL ) {
      printf( "symbol does not exist on the open map\n" );
   } else {
      printf( "The symbol with ID %d has the following values:\n",
               symbol_id );
      printf( "  variety: %s\n",
       ( p->symbol_variety == ovwIconSymbol ) ? "icon" : "connection" );
      printf( "  behavior: %s\n",
             ( p->behavior == ovwSymbolExecutable ) ?
               "executable" : "explodable" );
      printf( "  type: %s\n  label: %s\n",
             (char *) p->symbol_type, p->symbol_label );
      ...
      OVwFreeSymbolInfo( p );
   }
   OVwFreeMapInfo( map );
   ...
}
```

Using the OVwGetConnSymbol() Routine

The OVwGetConnSymbol() routine determines if a connection exists between any two icon symbols. If such a connection exists, the routine returns a pointer to an OVwSymbolInfo data structure containing complete connection symbol information. The routine has the function prototype:

```
OVwSymbolInfo *OVwGetConnSymbol( OVwMapInfo *mapInfo,
                        OVwSymbolId endpoint1, OVwSymbolId endpoint2 );
```

The following example shows how to list the connections of a metaconnection submap:

```
 ...
int i;
OVwSymbolInfo *syminfo;
OVwSymbolList *symlist;
OVwMapInfo *map = OVwGetMapInfo();
OVwSymbolId from_id, to_id;

syminfo = OVwGetConnSymbol(map, from_id, to_id);
if (!syminfo) {
   printf("No connection!\n");
} else if (!syminfo->var_un.conninfo.is_meta_conn) {
   printf("Single connection.\n");
   /* symbol represents the object syminfo->object */
} else {
   printf("Metaconnection.\n");
   /* get symbols on metaconnection submap */
   symlist = OVwListSymbols(map, syminfo->object.child_submap_id,
                            ovwAllPlanes, NULL);
   if (symlist) {
      for ( i=0; i<count; i++) {
        if (symlist->symbols[i].symbol_variety ==
                 ovwConnSymbol) {
            printf("Connection found.\n");
            /* symlist->symbols[i].object is object */
         }
      }
      OVwFreeSymbolList(symlist);
   }
}

if (syminfo)
   OVwFreeSymbolInfo(syminfo);
OVwFreeMapInfo( map );
 ...
```

You may specify the special value **ovwSubmapBackbone** for **endpoint2** if the layout algorithm of the submap is either bus or ring layout.

Using the OVwListSymbols() Routine

Applications frequently need to determine what symbols are present on a given submap. The OVwListSymbols() routine provides this service. You can retrieve a list of all symbols on the submap, or you can selectively retrieve a list of only those symbols for which your application has expressed interest or that are on a specific plane. The OVwListSymbols() routine has the function prototype:

```
OVwSymbolList *OVwListSymbols( OVwMapInfo *map, OVwSubmapId submapId,
                               OVwPlaneType plane, char *appName );
```

The **plane** argument may be **ovwAppPlane, ovwUserPlane,** or **ovwAllPlanes**. The **appName** argument can be either the application name or NULL. If **appName** contains the application name, OVwListSymbols() will return only those symbols for which the application has expressed interest. If **appName** is NULL, OVwListSymbols() will return a list of all symbols in the specified plane(s).

When finished, you should call OVwFreeSymbolList() to free the memory allocated by NetView for AIX. This routine has the function prototype:

```
void OVwFreeSymbolList( OVwSymbolList *symbolList );
```

Symbol Type Routines

The NetView for AIX program provides two routines that operate on symbol types. The OVwListSymbolTypes() routine retrieves a list of all the currently registered symbol types. This routine is useful for determining which symbol types are available. The OVwListSymbolTypeCaps() routine returns a list of the capabilities that would be set if the user were to add an object to the map using a specific symbol type. Refer to the man pages if you need more information on these routines.

Deleting Symbols

The NetView for AIX program provides two routines to delete symbols. The OVwDeleteSymbol() routine deletes a single symbol from the open map; the OVwDeleteSymbols() routine deletes a list of symbols. Either routine can be used to delete icon or connection symbols. These routines have the function prototypes:

```
int OVwDeleteSymbol( OVwMapInfo *mapInfo, OVwSymbolId symbolId );
int OVwDeleteSymbols( OVwMapInfo *mapInfo, OVwSymbolIdList
                      *symbolList );
```

In general, an application should only delete those symbols that it creates. Applications should not delete symbols created either by the user or by another application.

If you delete a symbol that is the last symbol on the map that represents an object, you might need to delete the object as well. "Deleting Objects" on page 97 describes how to delete objects. Chapter 10, "Map Events and Map Editing" on page 145 describes how to register for map editing events that indicate that an object should be deleted.

Chapter 9. Creating and Using Submaps

This chapter describes the use of maps and submaps in working with the NetView for AIX user interface. It explains how to create and manipulate submaps, how to retrieve submap information, and how to change the background of submaps.

To make effective use of the information in this chapter, you should be familiar with the relationships between objects, maps, submaps, and symbols. These concepts were introduced in “Understanding NetView for AIX Terms” on page 4. You should also be familiar with the routines described in Chapter 6, “Understanding the NetView for AIX User Interface” on page 61 and Chapter 7, “Creating and Using Objects and Fields” on page 79.

Understanding Maps

A map is a collection of NetView for AIX objects and their relationships. Users do not view maps directly; they view windows called submaps that display a subset of map information.

Among all the maps that might exist, users can select one map to be the open map. The open map is the only map that can be updated. If updates are needed for other maps, the updates are made at the time the map is opened. Only one map can be open at a time in any session. The submap routines described in this chapter operate on the open map.

Understanding Submaps

A submap is a collection of related symbols that are displayed in a single graphical window. A submap provides a view into the map object space. Each submap displays a different perspective of the information in the map, with the submaps typically organized in a hierarchical fashion.

The most common method users employ to navigate through submaps is double-clicking the mouse on special symbols called explodable symbols. Double-clicking on an explodable symbol will cause a submap to be displayed. The submap contains additional symbols that describe, in more detail, the object associated with the explodable symbol. The object associated with the explodable symbol is called the *parent object*. The submap that is displayed by double clicking on the symbol associated with the parent object is called a *child submap*. The child submap shows all the objects contained within the parent object.

The submap on which the parent object is represented is called the *parent submap*. Since several submaps may contain symbols that explode into the same submap, a submap may have several parent submaps.

For example, consider a submap that contains a single symbol that represents an entire organization. From this high-level view, a user could double-click on the symbol to display a child submap that contains a view of the next level of organization. From there, the user could select a specific department, followed by the selection of a specific node. Each of these submaps graphically displays a different map perspective.

Creating a Submap

From a programming point of view, the creation of a submap and the display of a submap are two separate operations. Applications can create submaps at any time and have them available to respond to a user request. Users control when the submap is displayed through the NetView for AIX graphical interface, by selecting a menu item or double-clicking on an explodable symbol.

Use the OVwCreateSubmap() routine to create a submap on the open map. It has the function prototype:

```
OVwSubmapId OVwCreateSubmap( OVwMapInfo *mapInfo,
        OVwObjectId parentObject,
        int submapPolicy, int submapType, char *submapName,
        int layout, unsigned int flags );
```

When calling the OVwCreateSubmap() routine, supply the following arguments:

mapInfo	points to an OVwMapInfo data structure that contains information about the map. Applications can retrieve map information in the OVwMapInfo data structure in two ways. Applications can call the OVwGetMapInfo() routine, or applications can save the map information that accompanies a map open event. Later sections will describe how to handle map open events and how to manipulate fields in the OVwMapInfo data structure. To create submaps, though, you can simply pass in the data structure returned by the OVwGetMapInfo() routine or supplied with the map open event. The example following this list will show how to use the OVwGetMapInfo() routine when creating submaps.
parentObject	the object ID of the parent object that this submap will represent. You can also specify **ovwNullObjectId**, which means that this submap does not have a parent object. This is known as an orphan submap.
submapPolicy	indicates whether this submap will be shared by other applications or if it will be owned exclusively by this application. The possible values are **ovwSharedSubmap** and **ovwExclusiveSubmap**.
submapType	an application-defined field that applications can use to classify submaps. You may use this field for any purpose that suits your application's requirements. If your application does not use a particular submap type, use the special value **ovwNoSubmapType**.
submapName	specifies the name of the submap. This name appears in the submap title bar. You should keep the name short, and the first letter of each word in the name should be capitalized.
layout	specifies the automatic layout algorithm that controls symbol placement on the submap. Choose one of the following:

- ovwNoLayout
- ovwPointToPointLayout
- ovwBusLayout
- ovwStarLayout
- ovwTreeLayout

- ovwRingLayout
- ovwRowColumnLayout
- ovwMultipleConnLayout

Submap layout algorithms are described in “Submap Layout Algorithms” on page 135.

flags	use either **ovwNoSubmapFlags** or **ovwDisableAutoLayout**. Coding **ovwDisableAutoLayout** initially disables automatic layout for the submap, but this setting can be changed by the user.

OVwCreateSubmap() checks the input arguments and, if all arguments are valid, creates the submap. It returns a submap ID that uniquely identifies the submap within the map. The submap ID is used in future calls involving the submap, such as adding symbols to the submap or deleting the submap.

The following example shows how to use OVwCreateSubmap() to create a shared submap. The submap has a bus symbol layout algorithm. We will assume that the object ID of the parent object is already known.

```
#include <OV/ovw.h>

OVwSubmapId sub_id;
OVwObjectId obj_id;
OVwMapInfo *map_info;

/* assume 'obj_id' is set to the ID of the parent object */
map_info = OVwGetMapInfo();
sub_id = OVwCreateSubmap( map_info, obj_id, ovwSharedSubmap,
     0,"Administer: System Mgmt", ovwBusLayout, ovwNoSubmapFlags );
if (OVwIsIdNull( sub_id ) ) {
   /* error processing */
}
```

At this point, the submap has been created and is internally known by the NetView for AIX program by the name returned in sub_id. No symbols have yet been placed on the submap, and the submap has not been displayed to the user.

Note: Many of the arguments to OVwCreateSubmap() are associated in some way with application style. Refer to the *NetView for AIX Application Interface Style Guide* for guidance on creating submaps.

Choosing When to Create a Submap

One of the important decisions you need to make as an NetView for AIX application developer is determining when to create submaps. Applications that are driven entirely by user requests (for example, menu item selection) may need to wait until the user selection has occurred before the submap can be created. Other applications that are driven by network or system events (for example, LAN monitoring applications) should create submaps as soon as possible.

In general, applications should create submaps as soon as they know that the submaps are needed (for example, when a new node or network is discovered). There are several advantages to creating submaps in this way:

- The map is always up to date and is a complete representation of the management environment.
- Users do not incur unnecessary delays when they display submaps.

- Status information is current for symbols with compound status source.
- Submap information is present in the event that a user saves a map snapshot.

Organizing Your Submap Hierarchy

If your application creates submaps, you should tie the submaps into an existing map hierarchy. There are two methods for doing this:

- You can create a submap hierarchy within your application and place a symbol representing a top-level object in the root submap. The symbol in the root submap provides an entry point into the application's hierarchy.
- You can create a child submap for an object in an existing submap hierarchy. If you want to add a child submap to an existing submap hierarchy but a prospective parent object is not already defined, you can add a parent object to a submap in the existing hierarchy.

Create a child submap of an existing compound object only if, from the user's point of view, it provides a meaningful extension to the existing hierarchy. For example, the submap of a computer system that displays the components in a computer system (software, peripherals, interface cards, etc.) is a good candidate for a child submap.

Refer to the *NetView for AIX Application Interface Style Guide* and the *NetView for AIX Administrator's Guide* for more information about submap hierarchies.

Special Submaps

There are some types of submaps that have special characteristics. These submaps are described here.

The Root Submap

Before you can create a hierarchy of submaps, one special submap must be chosen as the navigation entry point. This submap is called the *root submap*. The NetView for AIX program automatically creates a single root submap for each map. The root submap represents the highest, most abstract view of the map. You cannot delete a root submap.

The Home Submap

The NetView for AIX graphical interface lets users specify which submap they want to see when a map is opened. Rather than seeing the root submap, users can specify that they want to see some other submap. This is called the *home submap*. Selection of the home submap is done entirely through the NetView for AIX user interface. Applications do not need to know which submap is the home submap, nor is there any mechanism for programmatically determining the home submap for a map.

Orphan Submaps

Though most submaps are based on an object hierarchy, you can programmatically create a submap that is not related to a parent object. These submaps are called orphan submaps. Orphan submaps have limited value. The main drawback to using orphan submaps is that users cannot easily locate and display them through objects on the map. (Users can, however, display orphan submaps through a submap list box accessible through the main menu bar). In general, you should not create orphan submaps.

The Metaconnection Submap

Metaconnection submaps are submaps that the NetView for AIX program automatically creates to represent multiple connections between symbols. When a first connection is made between two symbols, the connection is treated as a simple connection. If additional connections are made between the symbols, the NetView for AIX program automatically creates a submap, called a metaconnection submap, that contains all the connection symbols. Users can view the metaconnection submap to see all connections between symbols. Developers cannot add connections directly to the metaconnection submap. Rather, connections are made between the symbols, and the NetView for AIX program automatically adds the new connections to the metaconnection submap.

Submap Layout Algorithms

When you create a submap, you also define the default algorithm used to place symbols on the submap. Layout algorithms are fixed for the life of the submap. Submaps can use any one of the following layout algorithms:

No Layout Algorithm	Use for unrelated objects or objects whose only relationship is having the same parent object. You can also use this layout algorithm for applications that need to specify the exact placement of symbols (for example, when using background graphics).
Point to Point	Use for point-to-point network connections, showing logical relationships between objects. This layout algorithm is also applicable for submaps whose objects have multiple arbitrary connections.
Bus	Use for network connections on a bus segment or for showing multiple objects that tie into a common relationship.
Star	Use for network connections on a star segment or for showing multiple objects connected into a common master or hub object.
Tree	Use to display nodes in a tree-like hierarchical view.
Ring	Use for network connections on a ring segment or for showing multiple objects with equal connectivity between all objects.
Row/Column	Use for submaps of unrelated objects, or objects whose only relationship is having the same parent object. This layout algorithm is preferred over no layout algorithm as it provides order for the user.
Multiple Connections	Use for submaps that primarily contain a set of connections between two objects (for example, the logical channels of a physical link).

The NetView for AIX graphical interface uses the submap layout algorithm to determine symbol placement within the submap. Though automatic placement is adequate for most applications, you can override the default symbol layout algorithm and place symbols at particular locations in the submap. This feature is described in “Changing a Symbol's Position” on page 122.

Submap Planes

When users look at a submap, they see a graphical window containing symbols, connections between the symbols, and, optionally, a special window background. But in fact, submaps are composed of three separate planes that are superimposed on one another. The NetView for AIX program manages the graphical information maintained in these planes. The three planes are:

- A background plane, which contains either a solid color or a user/developer supplied graphical image.
- An application plane, where applications add symbols.
- A user plane, where users add symbols.

The background plane is fairly straightforward. Applications can select a bit image to serve as a backdrop for the submap. The bit image resides in the background plane. If the background plane is empty, the NetView for AIX program displays a solid color pattern.

Symbols added by applications go on the submap application plane. The presence of a symbol on the application plane indicates that the underlying object has semantic meaning to the application.

Symbols added by users go on the user plane, at least initially. Applications can request notification whenever a user adds a symbol to the user plane. If the application judges that the symbol is appropriate for that submap, then the application can move the symbol from the user plane to the application plane, where it will be managed by the application. Applications should not normally add, modify, or delete symbols on the user plane.

Chapter 10, “Map Events and Map Editing” on page 145 describes how applications can determine when symbols or connections are added to submaps. See that chapter if your application needs to determine when icon or connection symbols are added to submaps.

Shared and Exclusive Submaps

When applications create submaps, they can specify whether the submap is a shared or an exclusive submap. A *shared* submap is one in which multiple applications use the same application plane. Any application can add, modify, or delete a symbol in a shared submap. Further, any application can delete a shared submap.

An *exclusive* submap is submap that can be modified only by the creating application or the end user. Only the creating application can add, delete, or modify symbols on the application plane of the submap. Users can add symbols only to the user plane of an exclusive submap. Only applications can create exclusive submaps. Submaps created by users through the NetView for AIX user interface are created as shared submaps.

In general, we recommend using shared submaps whenever possible. Refer to the *NetView for AIX Application Interface Style Guide* for more guidance on when to use shared versus exclusive submaps.

Displaying a Submap

There are two ways for submaps to be displayed, depending on how users interact with NetView for AIX:

- If the user performs any of the following steps, the NetView for AIX user interface automatically displays the submap.
  - The user double-clicks the mouse on an explodable symbol.
  - The user selects a submap through the NetView for AIX submap list box.
  - The user selects the "open" operation on a symbol's pop-up menu.

  If the user performs one of these steps and the submap does not already exist, the NetView for AIX program will notify the user of this and prompt for additional information needed to create the child submap. The NetView for AIX graphical interface will then display the submap. The application does not need to issue a call to the NetView for AIX API to display the submap.
- If a user requests an application action through an NetView for AIX menu item, the application must explicitly inform the NetView for AIX program when the submap should be displayed. For example, the user might be able to invoke an application-supplied action to create a submap. The submap might use the object selection list as a form of input. The application is responsible for explicitly telling the NetView for AIX program to display the submap.

To display a submap window, use the OVwDisplaySubmap() routine. The OVwDisplaySubmap() routine has the function prototype:

```
int OVwDisplaySubmap( OVwMapInfo *mapInfo,
                      OVwSubmapId submapId );
```

Given a submap ID, this routine does one of two things:

- If the submap is not already displayed, it displays the submap window.
- If the submap window is already displayed, it raises the window to the top.

Changing Submap Characteristics

When you create a submap using the OVwCreateSubmap() routine, you supply a number of arguments that define the submap's characteristics. These characteristics include the parent object, submap layout algorithm, submap policy, submap name, and submap type. Of these characteristics, the submap name is the only one that can be changed after the submap has been created. All other characteristics are fixed for the life of the submap.

To change a submap name, use the OVwSetSubmapName() routine. The routine has the function prototype:

```
int OVwSetSubmapName( OVwMapInfo *mapInfo,
                      OVwSubmapId submapId, char *submapName );
```

Supply a pointer to a mapInfo structure, the submap ID of the particular submap, and a character string containing the new submap name. The name does not have to be unique.

Deleting a Submap

The OVwDeleteSubmap() routine deletes a submap. You can delete any submap that has the shared submap policy, or you can delete an exclusive submap if your application created it. The OVwDeleteSubmap() routine has the function prototype:

```
int OVwDeleteSubmap( OVwMapInfo *mapInfo, OVwSubmapId submapId );
```

Caution: When you delete a submap, the NetView for AIX program assumes that all symbols on the submap should be deleted as well. The NetView for AIX program will delete all symbols on that submap. If a symbol is deleted that is the last symbol that represents an underlying object, then the object will be deleted also. Further, since an object might serve as a parent object of a child submap, deleting an object might result in the deletion of a child submap. Deleting a submap can set off a series of symbol, object, and submap deletions. This recursive deletion guarantees that unneeded symbols, objects, and submaps are removed when no longer needed. In general, applications should not delete submaps created by other applications. The OVwDeleteSubmap() routine does not delete the submap's parent object.

Getting Map and Submap Information

NetView for AIX provides a number of routines that are useful for finding out information about the open map and its component submaps. These routines can:

- Notify you when a map was created or last closed
- Notify you how the user has configured your application for a map
- Return a list of all submaps within a given map
- Retrieve submap information
- Produce a list of all objects on a map

These routines are described next.

Getting Map Information

The OVwGetMapInfo() routine retrieves information about the open map. It returns a pointer to a data structure containing the map name, map permissions, map creation time, the time the map was last closed, and the submap ID of the root submap. Map information is valid until the map is closed. The OVwGetMapInfo() routine has the function prototype:

```
OVwMapInfo *OVwGetMapInfo()
```

It returns a pointer to the OVwMapInfo data structure, which has the following form:

```
typedef struct {
   char *map_name;
   int permissions;          /* ovwMapReadOnly or ovwMapReadWrite */
   time_t creation_time;       /* map creation time */
   time_t last_closed_time;      /* time r-w map was last closed  */
   OVwSubmapId root_submap_id;     /* root submap for
                                      adding top-level symbols */
} OVwMapInfo;
```

The root_submap_id field is of particular interest, since it provides an entry point into the submap hierarchies present for the map. Using the OVwListSymbols()

routine you can retrieve a list of all symbols present in a given submap. Using the root_submap_id as an argument to this routine, you can determine which symbols, and hence, which submap hierarchies, are present in the root submap.

Many API routines take a pointer to the mapInfo data structure as the first argument. You should use the OVwGetMapInfo() routine to generate a mapInfo data structure. Though you could call OVwGetMapInfo() every time you need a mapInfo structure, a more efficient technique is retrieve the map information once and save it for future use. The most logical place to do this is in a callback routine that is registered for the ovwMapOpen event. Within that callback routine, you can use OVwCopyMapInfo() to save the map information returned with the ovwMapOpen event. The OVwCopyMapInfo() routine copies the memory in an OVwMapInfo data structure.

When finished with the OVwMapInfo data structure, call OVwFreeMapInfo() to free the memory allocated by NetView for AIX.

Getting Application Configuration Information

Users can configure the behavior of some applications through an application configuration dialog box that appears when either a new map is opened or the map configuration menu item is selected. User configuration may impact the application's initialization behavior. This section describes how applications can determine how they are configured by the user.

Some applications present an application configuration dialog box to the user when a new map is opened. The application configuration dialog box is defined in an Enroll block in an Application Registration File. The Enroll block contains entries for all configuration fields used by the application. (The *NetView for AIX Application Interface Style Guide* describes how applications should support application configuration). Applications can retrieve the values of application configuration fields using the OVwGetAppConfigValues() routine. OVwGetAppConfigValues() has the following function prototype:

```
OVwFieldBindList *OVwGetAppConfigValues(OVwMapInfo *map,
                                        char *appName );
```

OVwGetAppConfigValues() returns a pointer to a list of field values. The OVwFieldBindList data structure is described in “Getting a List of Object Fields” on page 94. By following the pointers in the field data structures, applications can determine the values of all application configuration fields. Based on these values, the application can determine how the user has configured the application. For example, the user may have disabled the application, or they may have enabled only certain parts of application functionality. The application, not NetView for AIX, is responsible for determining how the fields are used and what the field values mean. When finished, call OVwFreeFieldBindList() to free the memory allocated by the OVwGetAppConfigValues() routine. Applications can set the application configuration fields using the OVwSetAppConfigValues() routine. The OVwSetAppConfigValues() routine has the function prototype:

```
int OVwSetAppConfigValues( OVwMapInfo *map,
OVwFieldBindList *configParams );
```

This routine is useful for setting the values of general fields that may change. The default values for application configuration fields can be set in the configuration enroll block in the Application Registration File. See the man pages if you need more information on this call.

Getting Submap Information

The NetView for AIX EUI API provides two routines that retrieve information at the submap level, OVwGetSubmapInfo() and OVwListSubmaps().

Using OVwGetSubmapInfo()

The OVwGetSubmapInfo() routine returns complete information about a given submap, including the submap name, policy, parent object ID, etc. It has the function prototype:

```
OVwSubmapInfo *OVwGetSubmapInfo( OVwMapInfo *mapInfo,
               OVwSubmapId submapId );
```

It returns a pointer to an OVwSubmapInfo data structure. The data structure has the form:

```
typedef struct {
   OVwSubmapId submap_id;
   char *submap_name;
   int submap_policy;         /* ovwSharedSubmap, ovwExclusiveSubmap */
   char *app_name;            /* app that created submap;
                                 NULL = user-created */
   int submap_type;           /* application-specified value */
   OVwObjectId parent_object_id; /* parent object of submap;
                                    ovwNullObjectId if orphan   */
   int layout_style;          /* automatic layout algorithm */
   OVwBoolean layout_on;      /* has user disabled auto layout? */
   char *bg_graphics;         /* path of bitmap, NULL if none */
} OVwSubmapInfo;
```

When finished, call OVwFreeSubmapInfo() to free the memory allocated by OVwGetSubmapInfo().

Using OVwListSubmaps()

The OVwListSubmaps() routine retrieves a list of submaps on the open map. There are a variety of ways to filter which submaps are included in the list:

- You can select whether the list of submaps should include all submaps, or only those created by a particular application.
- You can filter the list of submaps to include only those submaps that have a particular submap type. Applications define submap types when they create submaps.
- You can filter the list of submaps to include only those submaps whose parent objects have particular field values. Field values are defined using an OVwFieldBindList structure, which is a list of field values. The OVwFieldBindList data structure is described in “Getting a List of Object Fields” on page 94.

The OVwListSubmaps() routine has the following function prototype:

```
OVwSubmapList *OVwListSubmaps( OVwMapInfo *mapInfo, char *appName,
              int submapType, OVwFieldBindList *parentFieldValues );
```

The arguments to OVwListSubmaps() are used in the following way.

mapInfo	A pointer to a map information data structure.
appName	If NULL, NetView for AIX will search all submaps on the open map. If it is not NULL, NetView for AIX will limit the search to submaps created by the application with the name appName.
submapType	A value that has meaning to the creating application. The special value **ovwAnySubmapType** matches any submap type. Submaps created by users always have a value of **ovwAnySubmapType.** Submap types were described in “Creating a Submap” on page 132.
parentFieldValues	A pointer to a OVwFieldBindList data structure that contains the particular field values against which parent objects should be compared. The OVwFieldBindList data structure is described in “Getting a List of Object Fields” on page 94.

The OVwListSubmaps() routine returns a pointer to an OVwSubmapList structure. The OVwSubmapList data structure uses NetView for AIX's standard list form: it contains a integer count of the number of entries in the list, and a pointer to the first in a contiguous array of entries.

The following code segment shows how to use the OVwListSubmaps() routine. This code retrieves a list of submaps that were created by any application and that have parent objects that have the isNode capability field set to TRUE.

```
#include <OV/ovw.h>

OVwSubmapList *list_ptr;
OVwMapInfo *mapInfo;
int i;
OVwFieldId f_id;
OVwFieldBinding binding;
OVwFieldBindList bind_list;
OVwFieldValue field_value;

f_id = OVwDbFieldNameToFieldId( "isNode" );

binding.field_id = f_id;
binding.field_val = &field_value;

binding.field_val->is_list = FALSE;
binding.field_val->field_type = ovwBooleanField;
binding.field_val->field_bool_val = TRUE;

bind_list.count = 1;
bind_list.fields = &binding;

mapInfo = OVwGetMapInfo();
list_ptr = OVwListSubmaps( mapInfo, NULL, ovwAnySubmapType,
                 &bind_list );
if ( ! list_ptr ) {
   /* error processing */
}

for ( i=0; i<count; i++ ) {
   printf( "%s\n", list_ptr->submaps[i].submap_name );
}
OVwFreeSubmapList( list_ptr );
OVwFreeMapInfo( map );
```

When you are done with the data structure returned by OVwListSubmaps(), call the OVwFreeSubmapList() routine to free the data structure memory allocated by NetView for AIX. OVwFreeSubmapList() has the function prototype:

```
void OVwFreeSubmapList( OVwSubmapList *submapList );
```

Using Submap Background Graphics

By default, the NetView for AIX program uses a solid color pattern as a background for all submaps. The background graphic for any submap can, however, be changed to contain a bit image. The background graphic can be changed either by the end user through the NetView for AIX user interface, or by the developer through the NetView for AIX EUI API. Background graphics provide a number of useful functions. You can, for instance, choose a more visually pleasing background with a different pattern or color. But the most useful function, by far, is the ability to display a graphical map of the environment being managed. For example, you can display a bit image of a country, a state, or an office or building floor plan. By placing symbols at coordinates relative to the background graphic, you can provide context for submap symbols in a meaningful way.

Setting and Clearing Background Graphics

The NetView for AIX EUI API provides two routines that let you programmatically control the background graphics for submaps. They have the following function prototypes:

```
int OVwSetBackgroundGraphic( OVwMapInfo *mapInfo,
                     OVwSubmapId submapId, char *filename );
int OVwClearBackgroundGraphic( OVwMapInfo *mapInfo,
                     OVwSubmapId submapId );
```

The OVwSetBackgroundGraphic() routine takes a submap ID and the full pathname of a bit image file, and it sets the background graphic for the specified submap. The NetView for AIX program scales the bit image to fit the window size. Use the OVwClearBackgroundGraphic() routine to restore the submap background to the default solid color pattern.

Note: The background manipulation routines can take several seconds to run.

Symbol Placement and Background Graphics

If you use a background graphic that provides some form of geographical map (for example, a building floor plan), then you will probably want to place symbols at particular locations relative to that map. For this placement to work correctly, specify **ovwNoLayout** as the layout algorithm when you create the submap.

Bit Image Formats

For your convenience, NetView for AIX lets you use background graphic bit images in either of two formats:

- The Graphics Interchange Format (GIF) developed by CompuServe.
- The X Bitmap format. See the bitmap() man page.

The NetView for AIX program handles the details of displaying different graphics image formats.

Chapter 10. Map Events and Map Editing

This chapter introduces the NetView for AIX map editing routines, which provide many functions for applications that interact with users who perform map-related operations such as adding objects, creating submaps, or deleting symbols. Read this chapter if you want your application to be notified when users or other applications modify the map. This chapter also provides information that you will need if your application supports any of the four map-editing operations: adding a symbol to a submap, connecting two symbols on a submap, modifying an object's attributes, or changing an application's configuration.

This chapter addresses these topics:

- Receiving notification of map changes
- Opening and closing maps
- Participating in map changes
- Cut and paste operations

Receiving Notification of Map Changes

Applications can register callback routines that are invoked when specific NetView for AIX events occur. See "NetView for AIX Events" on page 63 for a list of 31 NetView for AIX events. There are many other map-related events that applications might also find interesting. The NetView for AIX program generates an event when one of the following conditions occurs:

- The map is opened or closed.
- A symbol, object, or submap is created.
- A symbol, object, or submap is deleted.
- A symbol is moved.
- The selection list is changed.
- The status value of a symbol or object is changed.
- An object capability field is changed.
- The user indicates that an object should be managed or unmanaged.
- The user indicates that an object should be acknowledged or unacknowledged.
- A symbol is hidden or unhidden.

The next section describes the NetView for AIX map editing events. You can register your application to receive these events with the OVwAddCallback() routine.

Map Editing Events

Table 11 summarizes the map editing events. The left column contains the event types. The right column contains the event meaning and the name of the function prototype for the corresponding callback routine.

Table 11 (Page 1 of 2). Map Events and Callback Routines

Event Type	Meaning and Callback Routine
`ovwEndSession`	NetView for AIX terminated. `(*OVwEndSessionCB)(...)`
`ovwSelectionListChange`	The selection list changed. `(*OVwSelectListChangeCB)(...)`
`ovwMapOpen`	A map was opened. `(*OVwMapOpenCB)(...)`

Table 11 (Page 2 of 2). Map Events and Callback Routines

Event Type	Meaning and Callback Routine
`MapClose`	A map was closed. `(*OVwMapCloseCB)(...)`
`ovwConfirmDeleteSymbols`	One or more symbols were deleted from the map. `(*OVwConfirmDeleteSymbolsCB)(...)`
`ovwConfirmDeleteObjects`	One or more objects were deleted from the map. `(*OVwConfirmDeleteObjectsCB)(...)`
`ovwConfirmDeleteSubmaps`	One or more submaps were deleted from the map. `(*OVwConfirmDeleteSubmapsCB)(...)`
`ovwConfirmCreateSymbols`	One or more symbols were created on the map. `(*OVwConfirmCreateSymbolsCB)(...)`
`ovwConfirmCreateObjects`	One or more objects were created on the map. `(*OVwConfirmCreateObjectsCB)(...)`
`ovwConfirmCreateSubmaps`	One or more submaps were created on the map. `(*OVwConfirmCreateSubmapsCB)(...)`
`ovwConfirmMoveSymbol`	A symbol was moved within a submap. `(*OVwConfirmMoveSymbolCB)(...)`
`ovwConfirmManageObjects`	One or more objects became managed on the map. `(*OVwConfirmManageObjectsCB)(...)`
`ovwConfirmUnmanageObjects`	One or more objects became unmanaged on the map. `(*OVwConfirmUnmanageObjectsCB)(...)`
`ovwConfirmAcknowledgeObjects`	One or more objects became acknowledged on the map. `(*OVwConfirmAcknowledgeObjectsCB)(...)`
`ovwConfirmUnacknowledgeObjects`	One or more objects became unacknowledged on the map. `(*OVwConfirmUnacknowledgeObjectsCB)(...)`
`ovwConfirmExplodeObjects`	An object was exploded on the map. `(*OVwConfirmExplodeObjectsCB)(...)`
`ovwConfirmHideSymbols`	NetView for AIX made one or more symbols hidden. `(*OVwConfirmHideSymbolsCB)(...)`
`ovwConfirmUnhideSymbols`	NetView for AIX made one or more symbols unhidden. `(*OVwConfirmUnhideSymbolsCB)(...)`
`ovwConfirmSymbolStatusChange`	The status of one or more symbols changed. `(*OVwConfirmSymbolStatusChangeCB)(...)`
`ovwConfirmObjectStatusChange`	The object status of one or more objects changed. `(*OVwConfirmObjectStatusChangeCB)(...)`
`ovwConfirmCompoundStatusChange`	The compound status of one or more objects changed. `(*OVwConfirmCompoundStatusChangeCB)(...)`
`ovwConfirmCapabilityChange`	The capability field of one or more objects changed. `(*OVwConfirmCapabilityChangeCB)(...)`

The names of the callback routine function prototypes are derived from the event name. For example, the `ovwConfirmSymbolStatusChange` event has the callback routine (*OVwConfirmSymbolStatusChangeCB)(). This naming convention lets you easily translate between event types and function prototypes.

The concepts behind most of these events have been presented earlier in this manual. Object, symbol, and submap creation, deletion, and modification are described in previous chapters. You can refer to those chapters if you need to review those operations. There are, however, two new concepts that have not been described yet: hidden symbol events and unmanaged object events. These are described next.

Hidden Symbol Events

There are some situations in which the user might not want a symbol to appear in submaps, but, for some reason, the application will not allow the symbol to be deleted. In this case, the user can hide the symbol using the hide operation in the NetView for AIX graphical interface. Though the symbol still exists, it is no longer presented in the NetView for AIX graphical interface. When a symbol is hidden, the NetView for AIX program sends the `ovwConfirmHideSymbols` event to all applications that have registered a callback routine for that event.

A hidden symbol is not deleted. The symbol still exists and can be operated upon by applications just as if it were visible. Applications can still perform all symbol-related operations, such as modifying symbol attributes or retrieving symbol information. For instance, an application might be coded to ignore a hidden symbol when performing application-specific status propagation, because hidden symbols should not contribute to compound status. Users can request that a hidden symbol be made visible again. When a user selects this operation, the NetView for AIX program will send an `ovwConfirmUnhideSymbols` event to all registered applications.

Manage and Unmanage Events

Through the NetView for AIX graphical interface, users can control whether an object is managed or not. Symbols representing unmanaged objects do not receive status updates from applications. Applications will receive errors if they attempt to change the status on an unmanaged object or on the symbol that represents the unmanaged object. An unmanaged object remains unmanaged until the user explicitly re-enables management of the object through the NetView for AIX graphical interface.

All symbols representing an unmanaged object have the same status color to reflect the unmanaged status. By default, unmanaged icon symbols are wheat in color, and unmanaged connection symbols are black. See Chapter 8, “Creating and Using Symbols” on page 99 if you need more information about status colors and their meaning. Note that all symbols representing an object on a particular map, regardless of the symbol status source, are changed to unmanaged status if the underlying object is unmanaged. When an application is registered for the ovwConfirmUnmanageEvent event, it can determine if the object is currently managed on other maps. Applications can examine the `op_scope` field in the `OVwObjectInfo` data structure to determine whether the object is still managed on any other maps. The `op_scope` value will be `ovwAllMapsScope` if the object has been unmanaged on the last map on which it was managed. You can code your application to discontinue monitoring an object if it is no longer being managed on any map.

Similarly, your application can register for acknowledge and unacknowledge events to know when the user acknowledges or unacknowledges an object. When an object is changes from unmanaged to managed, or from acknowledged to unacknowledged, its status will be unknown, and you must determine how you want your application to deal with that object.

An Event Handling Example

The following example shows how you might define a callback routine to handle the creation of a symbol by another application:

```
#include <OV/ovw.h>

void *create_symbol_CB( userData, type, map, symbolList )
void *userData;
OVwEventType type;
OVwMapInfo *map;
OVwSymbolList *symbolList;
{
 ...
}

main()
{
   int ret;

   if ( OVwInit() < 0 ) {
      printf( "application couldn't connect to NetView for AIX\n" );
      exit( 1 );
   }

   ret = OVwAddCallback( ovwConfirmCreateSymbols, NULL,
                   (OVwCallbackProc) create_symbol_CB, NULL );
   if ( ret < 0 ) {
   /* error processing */
   }
   OVwMainLoop();
}
```

This example shows how an application passively awaits notification that an event has occurred. The application does not control whether the symbol creation is allowed; it is simply notified that the event occurred. The next section describes how applications can actively interact with the NetView for AIX program to control whether certain map-editing operations are allowed.

Opening and Closing Maps

If your application modifies the map, there are certain steps that it must perform whenever a map is opened or closed by the user. The following sections describe how applications should respond to user requests to open or close a map.

Processing a Map Open Request

You can request that your application be informed of any user request to open a map by registering a callback routine for the ovwMapOpen event. Within that callback routine, the application might need to perform these steps:

- Determine what changes, if any, are permitted for the map. The main factors in this decision are the map permissions and whether the user has enabled the application for the new map through the application configuration values.
- Make the updates to the map as appropriate for the application. Design your application to examine the time at which the map was last closed to determine what changes must be made to make the map current.

- If a significant number of time-consuming updates must be made, code your application to call special API routines to inform the user that a potentially lengthy map update is occurring. The process of updating a new map is referred to as *map synchronization*.

Receiving Map Open Events

Applications register for the **ovwMapOpen** event using the OVwAddCallback() routine. The map open callback routine has the following function prototype:

```
void (*OVwMapOpenCB)( void *userData,
                      OVwEventType type,
                      OVwMapInfo *map,
                      OVwFieldBindList *configParams );
```

The arguments to the callback routine are as follows:

userData	A pointer to data defined by the user when the callback routine was registered.
type	The event (ovwMapOpen in this example).
map	A pointer to a OVwMapInfo data structure. The *permissions* field and the *last_closed_time* field in the OVwMapInfo data structure are of special interest. Applications can check the permissions to determine whether all updates are permitted (read-write permission) or only status updates are permitted (read-only permission). The map close time is useful in determining what changes are needed to make the map current.
configParams	Used to return application configuration values. Applications can traverse the pointers in the **OVwFieldBindList** data structure to access values for each of the application configuration fields.

You can design your application to use the map information and application configuration parameters to determine whether updates are permitted, as well as what updates are required to make the map current. You can use the application configuration values in any way that is appropriate for your application. For example, some applications might use a boolean configuration field to indicate whether the application is enabled or disabled for the map. Other applications might use a numeric scheme where different values indicate that different levels of map updates are permitted. It is the developer's responsibility to determine how the user has configured the application and to act appropriately.

Synchronizing the Map

Anytime a new map is opened, there may be a brief period in which map applications are updating the map to reflect changes that occurred since the map was last used. You can inform users that the map is being updated, by having your application present a message stating that the map is being synchronized.

Map Synchronization Routines: The NetView for AIX program provides two routines that inform users of map synchronization. These routines are optional but recommended. They have the function prototypes:

```
int OVwBeginMapSync( OVwMapInfo *mapInfo );
int OVwEndMapSync( OVwMapInfo *mapInfo );
```

Call the OVwBeginMapSync() routine to mark the beginning of a synchronization phase. Other applications may also synchronize at the same time. While one or more applications are synchronizing, the user sees an appropriate message in the status line of all submap windows.

Each application should call the OVwEndMapSync() routine to mark the end of the synchronization phase. When the last map application completes synchronization, the synchronizing message is removed from the graphical interface. Users can then view submaps with confidence that the information is current. Because some user actions are inhibited during synchronization, be sure to call the OVwEndMapSync() routine so that these actions will be permitted.

Checking for Events while Synchronizing: Applications can manually check the NetView for AIX event queue for the presence of specific events that would justify processing interruption. Since map synchronization is an operation that can take a long time, consider having your application check the event queue for the presence of important events, such as the ovwMapClose event. Use the OVwPeekOVwEvent() routine to check the event queue for the presence of a particular event.

Starting an Application after a Map Is Opened

When the NetView for AIX program starts a new user session, the map is opened before the map applications are started. Applications do not receive an ovwMapOpen event if the map is opened before the application is started. Because of this behavior, applications need to take special steps at startup time to determine if the currently open map needs to be updated. These steps include retrieving the map information for permissions and last-close time and the application configuration values.

You can design your map-open callback routine to synchronize the map whether the application is being started or it is already running. You can manually retrieve the map information and application configuration values, and you can invoke your callback routine directly. This simulates the reception of an ovwMapOpen event. The following example illustrates this technique:

```
#include <OV/ovw.h>

void mapOpenCB( void *userData, OVwEventType type,
                OVwMapInfo *map_info,
                OVwFieldBindList *config_params )
{
/* check the map permissions, last close time, and
app configuration values to determine what map updates are needed */
OVwBeginMapSync( map_info );
OVwEndMapSync( map_info );
}

main()
{
   int ret;
   char *appname;
   OVwMapInfo *map;
   OVwFieldBindList *bind_list;

   ret = OVwInit();
   ret = OVwAddCallback( ovwMapOpen, NULL, (OVwCallbackProc)
                         mapOpenCB, NULL );
   /* Get the map information and app config values, then
invoke the callback routine */
   map = OVwGetMapInfo();
   appname = OVwGetAppName();
   bind_list = OVwGetAppConfigValues( map, appname );
   mapOpenCB( NULL, ovwMapOpen, map, bind_list );

   /* Free the map, bind_list, and appname  */
   OVwFreeMapInfo(map);
   OVwFreeFieldBindList(bind_list);
   free appname;

   /* start checking for events */
   OVwMainLoop();
}
```

Processing a Map Close Request

When the user requests to close a map, the NetView for AIX program begins a dialog with each application that has registered callback routines for the ovwMapClose event. The NetView for AIX program provides a number of arguments, including a proposed closing time for the map, to each application. Each application responds to the NetView for AIX program, indicating that the close request was acknowledged. Within that acknowledgment, the application can agree to the proposed close time, or it can reply with an earlier close time. An application can specify an earlier closing time if it is in the midst of updating a map. The NetView for AIX program closes the map when it receives acknowledgment from all applications registered to receive the close event. If applications do not respond within the configured time period, the NetView for AIX program assumes that application agrees with the proposed closing time, and proceeds with closing the map. The acknowledgment timer is configurable through the X resource file, /usr/OV/app-defaults/OVw.

NetView for AIX uses the earliest of all the returned map close times as the official map close time. The next time the map is opened, it will reflect the earliest close time returned by all applications. Applications can then make appropriate updates to make the map current.

Note: Applications should explicitly acknowledge the close event whenever possible. The NetView for AIX program will use a timer to detect unacknowledged close events, but that timer duration is typically quite long.

The map close event callback routine, OVwMapCloseCB(), and the map close acknowledgment routine, OVwAckMapClose, have the following function prototypes:

```
void (*OVwMapCloseCB)( void *userData, OVwEventType type,
                    OVwMapInfo *map, time_t closing_time );
int OVwAckMapClose( OVwMapInfo *map, time_t close_time );
```

See the man pages if you need more information about these routines.

Participating in Map Changes

In some cases, applications must cooperate with the NetView for AIX program to control whether map editing changes are allowed.

Map Editing Interactions with NetView for AIX

The NetView for AIX program attempts to establish a dialog with an application when the user performs one of the following NetView for AIX map operations:

- Adding a symbol
- Changing the application configuration for the map
- Connecting two symbols
- Modifying an object's attributes

These map operations require dialog boxes and are described in "Defining Dialog Boxes with the Enroll Block" on page 38.

When a user performs one of these NetView for AIX map editing operations, the NetView for AIX program displays a dialog box. The structure of the dialog box is defined by the Enroll block definitions in the application's ARF. The following steps occur:

Step 1. The user sets or changes the value of a field in a dialog box.

Step 2. The user selects the Verify button. The NetView for AIX program sends a query event to the responsible application to verify that the changes are acceptable. The application must have previously registered a callback routine, called a *query* callback routine, which determines whether the user's changes are valid.

Step 3. The application's query callback routine verifies that the entries in the dialog-box fields are valid. The application synchronously calls a NetView for AIX routine, called a *verify* routine, and passes a boolean value indicating whether the dialog box fields are acceptable. The application can also return a message that will be displayed in an Application Message field in the dialog box, to explain to the user why the entries were not accepted.

Note: Each value in a dialog box field is described by an OVwFieldValue data structure. See "The OVwFieldValue Data Structure" on page 88 for a description of this structure. For each field there is a boolean flag called **modified**. The value passed to your application with the query callback is in an unspecified state. Before calling the verify routine, your application must set this flag to True for any field whose value it changes.

Step 4. If the application accepts the entries, the NetView for AIX program may enable the OK button to permit the user to proceed. If the application does not accept the entries, the OK button remains disabled. The user should re-enter new values and start the process again. If the application accepts the fields, and the NetView for AIX program enables the OK button, the user can cancel the operation or press OK to proceed.

Step 5. If the user selects OK on the application-specific dialog box and on the main dialog box for the operation, the NetView for AIX program performs the operation. The NetView for AIX program sends a *confirm event* to the application, confirming that the operation was performed. The application callback routine that handles the confirm event is called the *confirm* callback routine.

This handshaking mechanism between the NetView for AIX program and the application is known as the *Query-Verify-Confirm sequence*. There is a separate Query-Verify-Confirm sequence for each of the four map operations.

Note: For this form of map editing to work correctly, an application must have: 1) defined one or more enroll blocks in the ARF, and 2) registered both Query and Confirm callback routines. If an enroll block does not exist, the NetView for AIX program cannot construct the application-specific dialog box. If the callback routines are not registered, the NetView for AIX program cannot communicate with the application.

Query-Verify-Confirm Routines

The following list contains the callback routine and verify routine function prototypes that apply for each of the four map operations. The function prototypes are grouped by map operation.

Adding a Symbol

```
void (*OVwQueryAddSymbolCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwSubmapInfo *submap,
                OVwFieldBindList *capabilityFields,
                OVwFieldBindList *dialogBoxFields );

int OVwVerifyAdd( OVwMapInfo *map, OVwFieldBindList *dialogBoxFields,
                OVwBoolean verified, OVwBoolean appPlane,
                char *errorMsg );

void (*OVwConfirmAddSymbolCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwSymbolInfo *symbol,
                OVwFieldBindList *capabilityFields,
                OVwFieldBindList *dialogBoxFields );
```

Changing the Application Configuration

```
void (*OVwQueryAppConfigCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwFieldBindList *configParms );

int OVwVerifyAppConfigChange( OVwMapInfo *map,
                OVwFieldBindList *configParms,
                OVwBoolean verified, char *errorMsg );

void (*OVwConfirmAppConfigCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwFieldBindList *configParams );
```

Connecting Two Symbols

```
void (*OVwQueryConnectSymbolsCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwSubmapInfo *submap,
                OVwObjectInfo *object1,
                OVwObjectInfo *object2,
                OVwFieldBindList *capabilityFields,
                OVwFieldBindList *dialogBoxFields );

int OVwVerifyConnect( OVwMapInfo *map, OVwObjectInfo *object1,
                OVwObjectInfo *object2,
                OVwFieldBindList *dialogBoxFields, OVwBoolean verified,
                OVwBoolean appPlane, char *errorMsg );

void (*OVwConfirmConnectSymbolsCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwSymbolInfo *symbol,
                OVwObjectInfo *object1,
                OVwObjectInfo *object2,
                OVwFieldBindList *capabilityFields,
                OVwFieldBindList *dialogBoxFields );
```

Changing the Object Description

```
void (*OVwQueryDescribeCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwObjectInfo *object,
                OVwFieldBindList *dialogBoxFields );

int OVwVerifyDescribeChange( OVwMapInfo *map, OVwObjectInfo *object,
                OVwFieldBindList *dialogBoxFields, OVwBoolean verified,
                char *errorMsg );

void (*OVwConfirmDescribeCB)( void *userData, OVwEventType type,
                OVwMapInfo *map, OVwObjectInfo *object,
                OVwFieldBindList *dialogBoxFields );
```

If your application supports any of these four map operations, create the appropriate enroll blocks in your application's ARF and register the appropriate callback routines.

Note: If you register your application for any of the query callback routines, your application must call the associated NetView for AIX verify routine as part of the query callback processing. Users may experience unnecessary delays if your application does not call the verify routine as part of query event processing. Refer to the man pages if you need more information about these routines.

The following short code segments show how to use the Query-Verify-Confirm routines. This example shows how an application might handle Describe operations. Assume the following entry is present in the application's Application Registration File:

```
Application "User Management" {
   ...
   Enroll Describe {
      if isUser {
         Field "User Name" {
            EditPolicy NoEdit;
         }
         Field "Default Shell" {
         }
         Field "User Information" {
         }
      }
   }
   ...
}
```

The next code segment registers callback routines to handle editing interaction with the NetView for AIX program. Note that the dialog-box fields are passed to the callback routines in `OVwFieldBindList` data structures, which are described in “Getting a List of Object Fields” on page 94. The fields supplied with the query and confirm events correspond to the fields enrolled in the application-specific dialog box.

```
#include <OV/ovw.h>

void *my_Describe_Query_CB( userData, eventType, map,
                            object, dialogBoxFields )
void *userData;
OVwEventType eventType;
OVwMapInfo *map;
OVwObjectInfo *object;
OVwFieldBindList *dialogBoxFields;
{
int ret;

/* Check the validity of the dialog box fields. Depending on the */
/* result, return either success or failure to calling program   */
if ( /* the operation is valid */ )
   ret = OVwVerifyDescribeChange( map, object, dialogBoxFields,
                                  TRUE, NULL );
else
   ret = OVwVerifyDescribeChange( map, object, dialogBoxFields, FALSE,
                                  "operation not valid" );
}

void *my_Describe_Confirm_CB( userData, eventType, map,
                              object, dialogBoxFields )
void *userData;
OVwEventType eventType;
OVwMapInfo *map;
OVwObjectInfo *object;
OVwFieldBindList *dialogBoxFields;
{
/* Operation completed successfully; perform any
application-specific operations */
}
```

```
main() {
int ret;

if ( OVwInit() < 0 ) {
   printf( "error connecting to NetView for AIX\n" );
   exit( 1 );
}

ret = OVwAddCallback( ovwQueryDescribeChange,
                    (OVwCallbackProc) my_Describe_Query_CB, NULL );
ret = OVwAddCallback( ovwConfirmDescribeChange,
                    (OVwCallbackProc) my_Describe_Confirm_CB, NULL );
OVwMainLoop();
}
```

Choosing Which Confirm Event to Use

There are two ways in which applications can receive confirmation of the Add or Connect map-editing operations. You can register your application for the specific `ovwConfirmAddSymbol` or `ovwConfirmConnectSymbol` events that correspond to the Query-Verify-Confirm routines, or you can register it for the more generic ovwConfirmCreateSymbols event. For example, to have your application notified that a symbol has been added to a submap, register either the (*OVwConfirmAddSymbolCB)() callback routine or the (*OVwConfirmCreateSymbolsCB)() callback routine. They are invoked through different events with different function prototypes and arguments.

Your choice of which confirm event to use is largely based on two factors: the way the symbol is created, and the arguments that are supplied to the callback routines. The generic ovwConfirmCreateSymbols event is generated regardless of how the symbol is created (by the user through the graphical interface or by another application). The same event is generated when either an icon or connection symbol is added. In contrast, the `ovwConfirmAddSymbol` and ovwConfirmConnectSymbol events are generated only when icon and connection symbols are created, respectively, by the user through the map-editing features of the NetView for AIX graphical interface. Another difference is that the `ovwConfirmCreateSymbols` event can be associated with the creation of multiple symbols. The `ovwConfirmAddSymbol` and ovwConfirmConnectSymbol events are only associated with a single symbol. The data returned to the callback routines differ between the generic ovwConfirmCreateSymbols event and the specific `ovwConfirmAddSymbol` and ovwConfirmConnectSymbol events. For example, compare the following function prototypes:

```
void (*OVwConfirmAddSymbolCB)( void *userData,
                OVwEventType type, OVwMapInfo *map,
                OVwSymbolInfo *symbol,
                OVwFieldBindList *capabilityFields,
                OVwFieldBindList *dialogBoxFields );

void (*OVwConfirmCreateSymbolsCB)( void *userData, OVwMapInfo *map,
                OVwEventType type, OVwSymbolList *symbolList
);
```

Note that the arguments to the (*OVwConfirmAddSymbolCB)() callback routine provide complete, detailed information about the single symbol affected by the operation. The callback routine also receives a pointer to the complete list of object database fields enrolled by the calling application. The more generic (*OVwConfirmCreateSymbolsCB)() routine contains a pointer to a symbol list, which the application must read to find out about the affected symbols. The generic (*OVwConfirmCreateSymbolsCB)() routine does not provide access to application-specific object database fields. It is the application's responsibility to get these values if they are needed.

Deleting a Symbol

Symbol deletion through the graphical interface uses the Query-Verify-Confirm transaction, though in a slightly different way. Unlike the dialog boxes presented in the other map operations, the delete symbol dialog box does not have a Verify button. The User presses the OK button to proceed, or the Cancel button to cancel the symbol deletion. If the user presses the OK button, the NetView for AIX program sends the `ovwQueryDeleteSymbol` event to all applications registered for the event. The applications then respond with a boolean value indicating whether the symbol can be deleted.

- If all of the registered applications approve, the symbol is deleted and the NetView for AIX program sends the ovwConfirmDeleteSymbol event to all applications registered for that event.
- If one or more applications do not approve, the NetView for AIX program does not delete the symbol. The NetView for AIX program informs the user that the symbol cannot be deleted. At that point, the user can choose to hide the symbol.

The symbol deletion Query-Verify-Confirm routines have the following function prototypes:

```
void (*OVwQueryDeleteSymbolsCB)( void *userData, OVwEventType type,
             OVwMapInfo *map, OVwSymbolVerifyList *symbolVerifyList );

int OVwVerifyDeleteSymbol( OVwMapInfo *map,
             OVwSymbolVerifyList *symbolVerifyList );

void (*OVwConfirmDeleteSymbolsCB)( void *userData,
             OVwEventType type,
             OVwMapInfo *info, OVwSymbolList *symbolList );
```

If your application creates symbols, consider whether it should handle symbol deletion. An application should disallow symbol deletion only if the symbol is required for correct operation.

Handling Cut and Paste Operations

Applications can receive various events when users perform cut and paste operations through the NetView for AIX graphical interface. Applications do not, however, receive special cut and paste events. Instead, applications receive the conventional `ovwQueryDeleteSymbols` and `ovwConfirmCreateSymbols` events. The cut-and-paste sequence appears to an application as unrelated symbol-delete and symbol-add events.

When the NetView for AIX program performs the cut operation, it enters into a Query-Verify-Confirm transaction sequence with registered applications. Depending on how the applications respond in their verify calls, the symbol may be deleted.

When the symbol is pasted, NetView for AIX sends an ovwConfirmCreateSymbols event to all applications registered for that event. The applications can examine the symbol and move the symbol from the user plane to the application plane by calling the OVwSetSymbolApp() routine.

Using delete and add events to represent cut and paste operations has implications for developers. An application that receives an ovwQueryDeleteSymbols event might need to buffer information about a deleted symbol, because the information might be needed later. Your application cannot tell if an ovwQueryDeleteSymbols event comes from a user request to delete a symbol or from the cut portion of a cut-and-paste sequence.

Not all applications will need to buffer information about the last symbol deleted. If your application does not distinguish between the paste and add operations, you might not need to buffer any information about the previous deletion. You might be able to treat the paste and add operations similarly.

If your application treats the paste operation and the add operations differently, you will probably need to buffer some information about the last symbol or symbols deleted. For instance, you could buffer the object ID related to the symbol, the submap ID of the submap from which the symbol was cut, or relationships to other symbols. Most applications should buffer at least the object ID. When an ovwConfirmCreateSymbols event arrives, your application can compare the object ID in the buffer to the object ID for the event. If the object IDs match, you can assume that the ovwConfirmCreateSymbols event actually represents a paste operation. Further, your application can compare the event request against other information that has been buffered, such as the previous submap type, to see if the paste operation is allowed.

Note: The NetView for AIX program assigns new symbol IDs to pasted symbols. This means that you cannot directly compare symbol IDs to determine whether delete and add operations are unrelated or if they are actually cut-and-paste operations. Further, comparing object IDs may not be adequate in some cases. If the user cuts multiple symbols that have the same underlying object, you may not be able to distinguish the symbols in the paste operation without some other contextual information. You may need to also save the submap ID or other unique application identifier.

Integrating and Documenting Your Map Application

Applications that operate on maps might create objects or submaps that have meaning to other map applications. Developers are encouraged to leverage existing map applications to build new ones.

If your application could potentially serve as an integration point for other applications, you should provide adequate instructions for other developers to follow to integrate their application with yours. For example, describe completely the following application elements:

- General map application information such as names or flags
- Registration files if you use them
- Fields if you create or require special fields
- Symbols and restrictions on their use

- Submap types and hierarchy
- How your application uses dialog boxes
- Dependencies on other applications

One possible place to document this information is in your application's man pages.

Part 3. Using the NetView for AIX Management APIs

Chapter 11. Understanding the NetView for AIX Management Environment

Part 2, "Working with the NetView for AIX User Interface" described numerous ways of customizing the graphical user interface of the NetView for AIX program. Part 3, "Using the NetView for AIX Management APIs" describes the management functions of the NetView for AIX program, and how you can tailor them to meet the requirements of your network. It provides information about the following topics:

- Understanding network management concepts
- Understanding managers and agents
- Using the NetView for AIX network management APIs
- Filtering network events
- Working with non-IP topologies

This chapter presents key terms and concepts of network management and the roles of programs called *managers* and *agents*. It describes the network management system's interactions with the network, and how developers control and augment those interactions. Subsequent chapters will provide more information about the network management API routines.

Defining Network Management Systems

The basic functions of a network management system are:

- To collect and store data about network conditions.
- To issue and respond to notifications of network conditions.
- To issue commands that cause actions at network nodes.

This list does not include presenting network data to the user; the management system stores the data in a format that enables the graphical user interface to present it when requested.

Specific terms for certain network management concepts are defined here; refer to them as you proceed through this section.

Agent An agent is the part of a management application that presents a view of a managed object, accepts requests, and issues responses and notifications. The agent role is that of a *responder*, except when issuing notifications.

Attribute An attribute is an item of information that describes some property of a managed object. An attribute has an associated value, which may be simple or complex. For example, one attribute of your RISC System/6000* workstation is its serial number; the value would probably be a simple integer value. Other attributes can have much more complex values.

Managed object
A managed object is a data representation of some real resource that can be managed, from the perspective of the manager. Every managed object is a member of some specific *object class*, whose members all share the same set of attributes, notifications (events), behavior, and management operations.

Management program
A management program consists of a cooperating set of programs that perform management activities. Such a management program is generally distributed across a computer network. Each part of the management program has certain responsibilities.

Manager A manager is that part of a management program that has responsibility for performing management activities. It typically issues requests on behalf of a human user, organizes the data from responses, and stores the data for access by a graphical user interface. The manager role is that of a *requester*, except when responding to notifications.

Object instance
An *object instance* is a specific member of an object class. An object instance is identified by a distinguishing attribute. For example, the RISC System/6000 has one attribute, the serial number, that is unique among all other RISC System/6000 workstations. This is its distinguishing attribute.

Requester
A *requester* is a proactive entity, which requests data and services from one or more responders. Most of the activities of a manager place it in the role of a requester; however, a manager takes the role of a responder when it (asynchronously) receives a notification.

Responder
A *responder* is a reactive entity, which responds to, or services, the asynchronous requests of one or more requesters. Most of the activities of an agent place it in the role of responder; however, an agent takes the role of a requester when it issues a notification.

Transaction
A *transaction* is a conclusive exchange of messages between a manager and an agent. This may require two messages, for example, a request from the manager and a response from the agent.

Network Management Protocols and APIs

There are two widely-accepted network management protocols:

- CMIP, the Common Management Information Protocol. This protocol was designed for managing OSI networks, but it is applied in networks other than OSI networks. The services defined for CMIP are known as the Common Management Information Services (CMIS). When CMIP is used over a TCP/IP protocol, the combination is called CMOT (CMIP over TCP/IP). The NetView for AIX program provides a CMOT implementation that is compliant with RFC 1085. For additional information about CMOT, refer to *RFC 1085*.
- SNMP, the Simple Network Management Protocol. This protocol was designed primarily for managing TCP/IP networks.

  SNMP is a transaction-oriented protocol that allows network elements, such as hosts, terminal servers, gateways, and management agents, to be queried directly. Because it has low network overhead when making such queries, it provides an inexpensive way of gathering network statistics. It is ideal for real-time monitoring and other management programs.

Refer to *ISO IS 9596-1, Common Management Information—Protocol Specification* and *RFC 1157, Simple Network Management Protocol (SNMP)* for a full description of CMIP and SNMP.

Creating Management Applications

The NetView for AIX program provides the following APIs for creating management applications:

- The NetView for AIX SNMP API

  This API provides routines for use with networks that use the SNMP protocol.

- The XMP and XOM APIs

  The XMP API enables you to build applications for networks that use either SNMP or CMOT. XMP uses the XOM API for data management.

- The NetView for AIX GTM API

  This API enables you to monitor networks that do not use the Internet Protocol (IP). It also enables you to create a layered view of your network.

- The NetView for AIX filter API

  This API enables you to create and modify event filters within your application.

This chapter refers occasionally to these APIs; additional information is provided in subsequent chapters.

Using APIs in a Client/Server Environment

The majority of NetView for AIX APIs have been enabled for a client/server environment. Existing applications that reference the OVw, OVsnmp, nvSnmp, and XMP APIs do not need to know if they are running on a client or server. If your applications use APIs that require communication with a NetView for AIX daemon, either on a client where an application is running or a remote server, it is transparent to the application whether the communication is staying on the client or going to a server. Specifically, the enablement is as follows:

- The OVw API allows for a remote connection to the ovwdb daemon.
- The OVsnmp API allows for connecting to trapd to receive traps.
- The XMP API allows applications to connect to PMD.
- The SNMP API allows for receiving filtered traps on a client.
- The OVuTL API allows nettl APIs on the client.
- The event filtering API allows for processing of filters on a client.

The OVsPMD API for communication with the process management daemon (ovspmd) is not available on clients.

The semantics of network connections have been embedded into the details of the APIs. OVsnmp requests issued from a client do not first go through the server before going to the network. This has the following implications:

- If a network device is configured to support SNMP communication from a specific network management station, a client's IP address must be added to that network device's configuration. Otherwise, SNMP requests from the MIB browser, xnmgraph, and other SNMP-based applications will time out.

- By default, traps sent to loopback (127.0.0.1) on a client machine are lost. There is no trap reception mechanism on a client machine. Thus, all applications running on a client should send traps to the NetView for AIX server.

If you have a distributed application that has graphical applications running on clients and daemons running on the server, you may need to use two new APIs. Use the new OVDefaultServerName() routine to determine the name of the server machine. A call to this routine checks to see if there is a /usr/OV/databases/servername file on the machine. If the file exists, the machine is a client, and the contents of the file specify the server's hostname. If the file does not exist, NetView for AIX assumes the machine is a server and returns the current hostname.

Applications can use the NVisClient() routine to determine if the target machine is a client or server. This API can be used to determine the most effective communication mechanism for an application to use for connecting to its daemons. Applications on a client could use RPC or network sockets; an application on a server could use IPC mechanisms for improved performance.

Understanding Managers and Agents

Because the environment to be managed is distributed, the allocation of management activities is also distributed. This idea is crucial, and it is the basis for the concepts of managers and agents, which were defined in “Defining Network Management Systems” on page 167. Managers and agents distribute management activities throughout the managed network.

Note that these definitions express how agents and managers interact without limiting these interactions. An agent can respond to the requests of several managers, and a manager can request the services of several agents.

New managers and agents can be added, either as totally new processes or as replacements for old managers and agents, without significant impact on the overall system. The advantage of this modular approach is most obvious and useful where common services, likely to be used by many managers, are implemented in agents. In this case, new management programs do not have to provide these services; instead, they can use the already defined and existing agents.

Manager Functions

A manager is a proactive entity that acts as the tool of a user to obtain, analyze, present, and act upon information about the distributed system environment. A manager performs the following tasks:

- Collects and processes raw data.
- Stores data to be presented to the user through the graphical user interface.
- Obtains user requests and acts upon them. User requests might involve changing certain elements of the environment or gathering additional data.

A manager is typically driven by the requests of its user. However, many managers engage in “hidden” activity for data gathering and automated control.

In addition to their proactive role, most managers are alert to notifications that indicate a change in status somewhere in the environment. Although most of the activities of a manager place it in the role of a requester, it acts as a responder when it receives asynchronous notifications.

Managers in the NetView for AIX environment respond to user actions and other asynchronous events. The technique used to handle these asynchronous events depends on the API used to construct the manager. When using SNMP, you must register an appropriate callback function to handle inbound asynchronous messages. This callback determines the nature of the inbound message, which must be either a response to an earlier asynchronous request or an incoming notification. The callback function then directs control to a routine that responds appropriately to the message. The XMP API, on the other hand, does not use callback routines.

Some user actions can require that your management program call appropriate request functions to retrieve information from, or cause some action on the part of, a managed object. The associated callback for each user action must contain the code necessary to accomplish the user's request.

Managers do not typically generate notifications. This is a direct reflection of the OSI model of management. There is no physical barrier to prevent a manager from issuing notifications, but corruption of the model can cause undue complexity in the implementation.

Finally, managers exist to service users' requests. In the absence of a user, some managers can shut down without negatively affecting any other element of the distributed network management solution. Some managers, however, may need to remain running if their purpose is to monitor, record, or automatically react to network activity.

Agent Functions

An agent is a reactive entity, which waits for and services the requests of one or more managers. It is a manager's window into the management aspects of the managed object it represents. Agents can also service other agents.

Agents are fundamentally request-driven, responding to requests from managers. When an agent receives a request, it needs to invoke the appropriate routines to handle that request.

In addition to their reactive role, most agents perform a proactive role. Agents are responsible for flagging special events that occur in the objects they manage and for generating notifications. These notifications (event reports) receive special treatment and are relayed to any manager that has expressed interest in them.

Three basic concepts should be understood about the function of an agent:

1. An agent acts as a server.

   To build an object-oriented environment, an agent should be a server. A server simply responds to requests for its services and does nothing more. An agent should, for the most part, be completely driven by requests from other management programs. The exception is the requirement that the agent must be able to issue notifications to other management programs.

2. The externalized function of an agent is completely specified by the object definition.

   An agent responds to requests directed at any object instance without knowing who the requester is. The agent's response is based on the defined object class template. The goal of an agent is to support the object definition. As a developer, you need to develop software to carry out the requests that your agent supports and the notifications that are generated by the agent. Note that the object definition may be revised as you develop the agent.

   It is important to remember that all functionality in an agent should support the definition of the managed object. Local, internal management functions can be embedded in the agent; however, the only functions that are exposed by the agent must be those expressed in the object definition. Resist the temptation to let features beyond this scope intrude on your design or to redefine the object to account for such features.

3. An agent can either act as an interface to an existing object or maintain the object.

   The object can be any logical or physical resource on a network. The resource to be managed can exist independently of the agent. Thus, there are two types of agents:

   - An interface to some existing physical object, such as a modem.
   - An agent that maintains and represents a logical object.

The Structure of an Agent

An agent's structure has two parts:

- One part participates in communications with managers, accepting requests and issuing notifications.
- The other part accesses the managed object to carry out the requests and to detect the conditions that warrant issuing an event report.

After initialization, most agents are organized around a loop, waiting for one of the following circumstances:

- An incoming request. The agent first determines what the request requires, then acts on it. This can be simple or complex, depending on the agent and the request. After servicing the request, the agent re-enters the wait loop.
- Conditions in the managed object that warrant issuing an event report. The agent assembles the required information, issues the event report, and returns to the wait loop.

For each agent that you create, you should create a local registration file (LRF). The LRF identifies the agent to the NetView for AIX program and contains information about the agent and its managed object. See “The Local Registration File” on page 19 for more information on local registration files.

Accessing the Managed Object

To support the managed object you have defined, your agent must access that resource. You should build a dedicated access module that shields the complexity of accessing the resource from the rest of the code.

The following three techniques can be used to access objects:

- Polling
- Using other agents
- Using specialized access mechanisms

Polling: This technique polls the object for the necessary data. This means accessing your object at some interval to determine whether any thresholds have been exceeded or if some status has changed. You may also want to access the object in order to update your local copy of its data.

To do polling, set up a time-out from the wait loop in the main routine. For example, if you are using the XMP API, the mp_wait() function call enables you to specify a number of bound-session objects to wait on, and a time-out value. When the mp_wait() function call exceeds the time-out value, you can call a time-out handler.

Using Other Agents: Another technique is to access a managed object through another agent. Such an agent must be able to issue requests. This ability is the defining characteristic of an object manager. The processing associated with issuing requests is identical, whether it is done by an object manager or a manager.

Always give high priority to clearing incoming messages from the XMP queue in order to avoid having the socket to XMP overflow and lose data. You can use either asynchronous or synchronous calls to clear the XMP queue. Both synchronous and asynchronous calls remove the messages from the socket and queue the data for subsequent asynchronous retrieval.

Using Specialized Access Mechanisms: You can also access an object by communicating with specialized software or hardware available for that purpose. Your main routine could receive either a signal, a message, or an interprocess communication (IPC) message from a resource that has such an access mechanism.

Using Agents as Object Managers

Depending on the complexity of your network, you may see, and write, different kinds of agents. For example, an object manager is an agent that has characteristics of both an agent and a manager, except that it typically does not have the manager's user interface component. The object manager provides higher-level services than an agent alone could offer to the resources it manages. Object managers use the services of the simple agents just as any other manager would. In turn, the manager uses the services of the object manager, just as it would use the services of any other agent.

The following list describes two kinds of object managers:

Proxy Agent

A proxy agent is a dedicated translator for a resource whose actual agent uses a foreign protocol. A proxy takes requests in one language, such as CMIS, translates them to the foreign language (which is arbitrary), and submits the translated request to the actual agent. Any response from the actual agent is picked up by the proxy agent, translated into the language of the request (CMIS in this case), and resent to the manager.

You can use a proxy agent to allow SNMP access to nodes that do not support SNMP. When you configure a proxy, the proxy agent receives the SNMP request and forwards it to the requested node using whichever non-SNMP protocol the requested node supports.

Mediation Device
A mediation device is an object manager whose managed object is typically a large subnet. The mediation device represents that subnet to a higher-level manager. Mediation devices usually appear only in very large networks. They can include the translation function of a proxy agent.

The function of a mediation device is to distill the massive quantities of information available on subnets, extract the most relevant data, and make that data available to a higher-level manager.

Programming with Managers and Agents

In the NetView for AIX environment, a management program is split into object-related components (agents) and user-related components (managers).

Network elements, such as bridges, routers, and modems, are typically managed by agents that are developed specifically for those elements. Agents take care of management interactions for these objects through standardized services that hide the complexity of direct communication.

It is important, in an object-oriented environment, to distribute the functions of a management program appropriately between managers and agents. Table 12 provides guidelines for the division of functions:

Table 12. Division of Functions between Managers and Agents

Element	Description
Manager	• Generally has a user interface • Uses the services of one or more agents • Receives notifications • Directs agent activities • Is not required to be running all of the time
Agent	• Represents information and functions of possible interest to one or more managers regarding a set of objects • Is usually running • Does not interface with users • Interfaces with some types of objects • Generates notifications • Responds to manager requests

Transactions between Managers and Agents

A requester can send several types of messages to a responder. These messages are described in Table 13 on page 175.

Table 13. Types of Request Messages Used in Communications

Message Type	Valid Services	Description
Get	CMIS, SNMP	Obtain a value maintained by the managed object.
Set	CMIS, SNMP	Set a value maintained by the managed object.
Event-report	CMIS, SNMP	Report special conditions about a managed object.
Cancel-get	CMIS	Cancel a pending get request.
Get-next	SNMP	Obtain the name and value of the next SNMP variable in the object.
Action	CMIS	Invoke one of the actions defined for the managed object.
Create	CMIS	Create an instance of an object class.
Delete	CMIS	Delete an instance of an object class.

Each request message has a parameter to identify the target objects, and other parameters as required by its function and by the protocol used. Each request message also has a corresponding response message with necessary parameters and defined error values to accommodate failures.

For most of the requests, the nature of the operation is implicit in the function. The CMIS action operation request, however, is a generic request. Objects can be defined to include operations outside the scope of the other requests; the action operation request provides access to such operations.

SNMP Traps and CMIS Notifications

Managed objects can issue unsolicited messages, generically called *event reports*. In SNMP, such a message is called a *trap*; in CMIS, the term used is *notification*. One of the defining characteristics of a managed object is the event reports it can issue.

The event-report request is unique in that its purpose is not to elicit a reaction from a managed object, but rather to report special or abnormal conditions for the managed object. The event-report request is used by agents.

Figure 7 on page 176 represents the interactions between a manager and an agent.

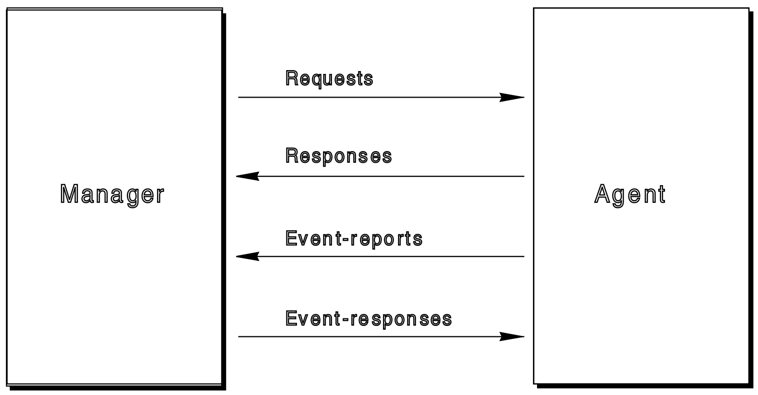

Figure 7. Interactions between Managers and Agents

An agent can both generate and receive event reports. For example, suppose an agent generates a notification because it detects a change of behavior in the object that could affect the network. You can code a higher-level agent to detect certain patterns or thresholds of these notifications and issue notifications based on these patterns or thresholds.

Characteristics of Transactions

Requests (including event-report requests) and responses can include parameters. In the case of requests, the parameters are input data; in the case of responses, they contain output data and error values.

You can further categorize transactions as either attribute-oriented or object-oriented. The attribute-oriented transactions operate on one or more attributes of the target object. These include the get, set, and get-next requests. The object-oriented transactions, create and delete, operate on the entire target object.

Two types of transactions, the action request and event-report request, cannot be strictly classified as either attribute-oriented or object-oriented. Actions can be defined in such a way that they may fall into either category. Notifications (event-reports) are not targeted at a managed object and cannot be classified as attribute-oriented or object-oriented. Notifications are directed to a manager.

Some requests require a response; others do not. The first case is called a *confirmed* request; the second case is called an *unconfirmed* request. For a confirmed request, the service invoker expects a reply from the service performer that contains information about the status of the request. For an unconfirmed request, lower layers of the protocol guarantee delivery of the request, but that is the end of the transaction. No reply is expected for an unconfirmed request.

The Communications Infrastructure

The communications infrastructure is the means by which messages are exchanged. XMP-based management programs automatically use the communications infrastructure. The communications infrastructure provides location transparency, a feature that enables managers and agents to access objects and agents without using hard-coded addresses. It consists of the following components:

- The pmd daemon
- The orsd daemon
- The ORS database

The pmd daemon directs management information between multiple managers and agents running concurrently. It is a message switch, determining its routing action either from user-specified addresses or from routing tables configured through the orsd daemon.

The ORS Database: The orsd daemon creates and maintains a global directory of agents, their locations, their protocols, and the objects each agent manages. Each agent should have that information registered with the orsd daemon by means of a local registration file. The orsd daemon permits dynamic modification of object-registration information. The orsd daemon ensures that all NetView for AIX nodes within the management domain are automatically updated as information changes. This directory is the object registration service (ORS) database.

The pmd daemon responds to all service needs, including association management and automatic retries of requests. An *association* is the relationship between a manager and another manager or agent that permits the exchange of information. XMP-based management programs do not have to initiate or control CMIS associations to the destinations of management requests.

For CMIS requests, the pmd daemon provides transparent control and initiation of all associations. SNMP is, of course, connectionless. For SNMP requests, the communications infrastructure hides the details of managing time-outs and retries.

Routing Messages: The communications infrastructure provides object location transparency. This means that you can specify an object instance for some operation without specifying the agent that is to perform the operation. For location transparency to work, the object and its agent must be registered with the communications infrastructure through the ORS.

If, for some request, you specify only the class and name of an object instance, the pmd daemon uses the database created by ORS to route the requested operation to the agent responsible for that object instance.

Usually, full object names are looked up in a database rather than keyed in by a user. Therefore, you can make requests and know that the name of the managed object contains sufficient information for the pmd daemon to route the request to the proper agent. You do not need to separately determine the address of the agent and supply that address to each request.

Note: There is a trade-off between the convenience of location transparency and network performance. Routing is most efficient if the distribution address of the agent is specified in each request. This avoids a search through the object registration data. The pmd daemon can supply the missing information and provide object location transparency, but there is a cost in performance.

Notifications, sometimes called event reports or traps, are a special case. Notifications are always routed through the event management service because the same notification might be reported to several management programs.

The following pages refer specifically to the OSI management environment. If you plan to use only the SNMP API, and are not interested in using the XMP API to develop CMOT-based network applications, you can skip ahead to “Developing Network Management Applications” on page 187.

Open Systems Interconnection (OSI) Management

The design of the NetView for AIX program is partially derived from the management model defined by the Open Systems Interconnection (OSI) subcommittee of the International Standards Organization (ISO). This model is called the OSI management model and defines the Common Management Information Services (CMIS) and Common Management Information Protocol (CMIP). The NetView for AIX program extends the OSI model to include networks that use Transmission Control Protocol/Internet Protocol (TCP/IP).

Object-Orientation in the OSI Model

The OSI management model uses an object-oriented approach, modelling all the system resources to be managed as *managed objects*. These resources include communications hardware, such as modems, bridges, and gateways, and software, such as operating systems, databases, and LAN programs.

Communications services, including OSI CMIP and Internet TCP/IP, were developed to allow communication between systems. However, standard communication protocols alone are not sufficient to manage complex distributed system environments. Distributed management programs require standard descriptions of the resources to be managed.

Therefore, the OSI management standards adopted an object-oriented model for encapsulating these resources, and for standardizing the interfaces they present to the network.

Using of the object-oriented model in management programs promotes the following:

Modularity	Functions are isolated into clearly defined and readily implemented modules.
Extensibility	Functions and services of existing management programs can be used as the foundations of new distributed management programs.

In object-oriented programming, code and data are fused into a single entity called a *managed object*. A managed object is composed of a set of attributes that characterize the object, actions that the object can perform, and notifications (messages) that the object can issue spontaneously.

A managed object represents a resource, such as hardware or software, that can perform operations or that can be monitored by a management program. The interface to the managed object is a management program called an *agent*. To request some action from a managed object, management programs send messages to the agent that represents the managed object. The managed object can respond to requests by returning a message. In fact, the only interaction between the manager and the managed object occurs through messages. Data about the system resource is encapsulated in the managed object, and that data is never manipulated directly by any other management program.

The object-oriented model naturally supports asynchronous processing. Asynchronous processing allows a management program to send a request message to a managed object and continue processing, perhaps corresponding with other objects, while waiting for a response from the initial request.

Understanding Objects

A managed object is an abstraction that represents a resource. The resource is an entity that is available somewhere on the network and has the capacity for being managed. The resource can be physical, such as a router or workstation, or it can be logical, such as a file system or a user. Examples of managed objects might include:

- A computer workstation
- A network interface card attached to a computer workstation
- A collection of free disk space data for nodes on a network

Those attributes of a resource that are related to the management of that resource are accessible through the managed object. Attributes that can not be managed are not included in the managed object definition. For example, some of the management attributes of a printer include its model, serial number, location, and paper supply status. A printer also supports operations that can be managed, such as reset and self-test. An example of a printer attribute that would not be part of a managed-object abstraction is the color of the printer. The color of the printer is an example of an attribute that cannot be identified, monitored, or requested to perform some operation programmatically.

Thus, as previously mentioned, managed objects are defined based on the following:

Attributes Items of information about the object.

For example, a RISC System/6000 workstation has attributes that include the amount of memory, disk size, type of network adapter, and serial number.

Actions The operations that the object can perform.

For example, a RISC System/6000 workstation supports startup and shutdown functions.

Notifications The messages that can be issued to notify other objects or management programs in the distributed system of conditions in the managed object.

For example, a RISC System/6000 workstation might issue a notification whenever a fatal system error occurs.

Understanding Object Classes

A *class*, which is also referred to as an *object class*, represents one or more objects that have common attributes. There are many objects that have the same attributes. For example, there may be several hundred RISC System/6000 workstations on a network. Each workstation is an instance of an object class. The collection of RISC System/6000 workstations (that is, the collection of the objects that have common attributes) represents an object class.

An object class is a description of the resource management properties that members of the object class have in common. The object class can be thought of as a logical structure that describes how each member of the class looks and behaves.

Objects Are Instances of an Object Class

An instance is a specific object in an object class. Thus, the term *instance* has the same meaning as the term *object*. Instances of a specific object class have the same mandatory package of attributes; however, the actual values of the attributes vary from instance to instance. For example, one attribute of the RISC System/6000 workstation class might be the size of the hard disk. One instance of that class (that is, one workstation) might have a 120MB hard disk, and another instance (a different workstation) might have a 200MB hard disk.

The Distinguishing Attribute

For each object class, one attribute is singled out as the key attribute that distinguishes each instance from all other members of that class. This attribute is called the *distinguishing attribute*.

For instance, a RISC System/6000 workstation class might have the following attributes:

- Amount of memory
- Size of hard disk
- Serial number
- Type of network adapter

All instances of this class, that is, all RISC System/6000 workstations, have these attributes. To identify one instance of the class (one RISC System/6000 workstation) from all other instances, one of the attributes must always have a different value for each instance. The serial number attribute must always have a different value for each RISC System/6000 workstation. Thus, the serial number attribute is the distinguishing attribute.

The Inheritance Relationships among Object Instances and Classes

In the object-oriented model, *inheritance* is the property by which one object class is defined as an extension of another. The new class has all the properties of the parent, plus any additional properties required.

The following three relationships provide a structured framework for defining and using objects:

- The registration relationship
- The containment relationship
- The inheritance relationship

Each of these relationships is hierarchical and is represented as a tree. They constitute a well-known base in the network and systems management community. While these relationships are not inherently part of the object-oriented approach, understanding them is important when you work with classes of objects in a network management system.

The Registration Relationship

The *registration relationship* is a hierarchy of unique identifiers for object classes, name bindings, actions, notifications, parameters, and attributes. The registration relationship pertains to these elements, not to object instances. Figure 8 shows part of a registration tree.

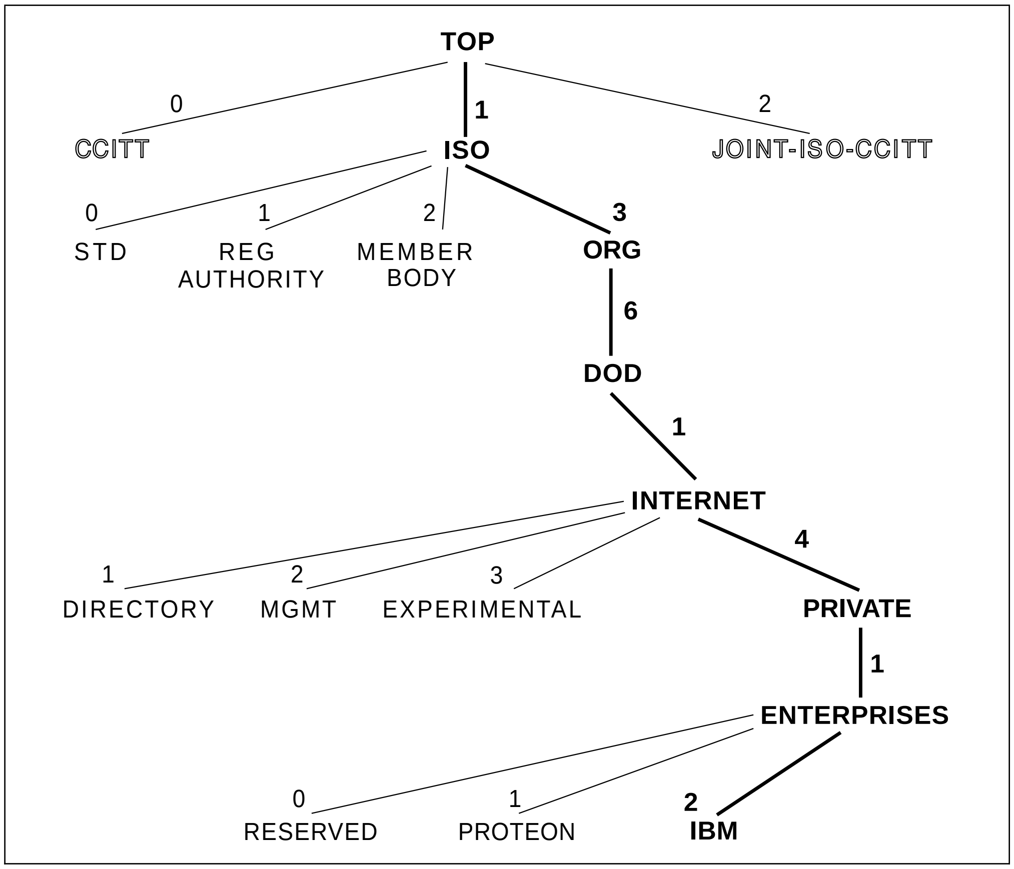

Figure 8. A Registration Tree

As object definitions are developed by businesses or organizations in the network community, they are assigned an identification number that uniquely identifies the business or organization and the object definitions. This is illustrated in Figure 8.

An object class identifier (OID) is the registration number that is made of a series of integers that traverse a path from the root of the registration tree to the object to be registered. Each branch of the registration tree has an associated registration authority that determines how numbers in the subtree beneath it are allocated. For example, IBM is a branch derived from the private enterprise branch of the registration tree. The path, 1.3.6.1.4.1.2, defines a branch for which IBM is the registration authority.

The registration relationship must be established during object definition. The registration identification is used during design to ensure that object classes, attributes, actions and notifications are uniquely identified within the registration tree. Management programs require correct registration identification numbers to identify the unique class of a managed-object instance.

Developers use documents published by ISO to identify existing object definition and registration relationships. The existing object definitions are often referred to as a *management information base* (MIB). Registration numbers are also used by the management program to identify the class of a managed-object instance.

The Containment Relationship

The containment relationship describes how each object instance is related to other object instances within a particular environment. This relationship pertains to object *instances*, not object *classes*. The containment relationship defines OSI addressing and is not associated with IP or SNMP environments.

All object instances are contained within other object instances. The containing object instance is called the *superior* object instance and the contained object instance is called the *subordinate*.

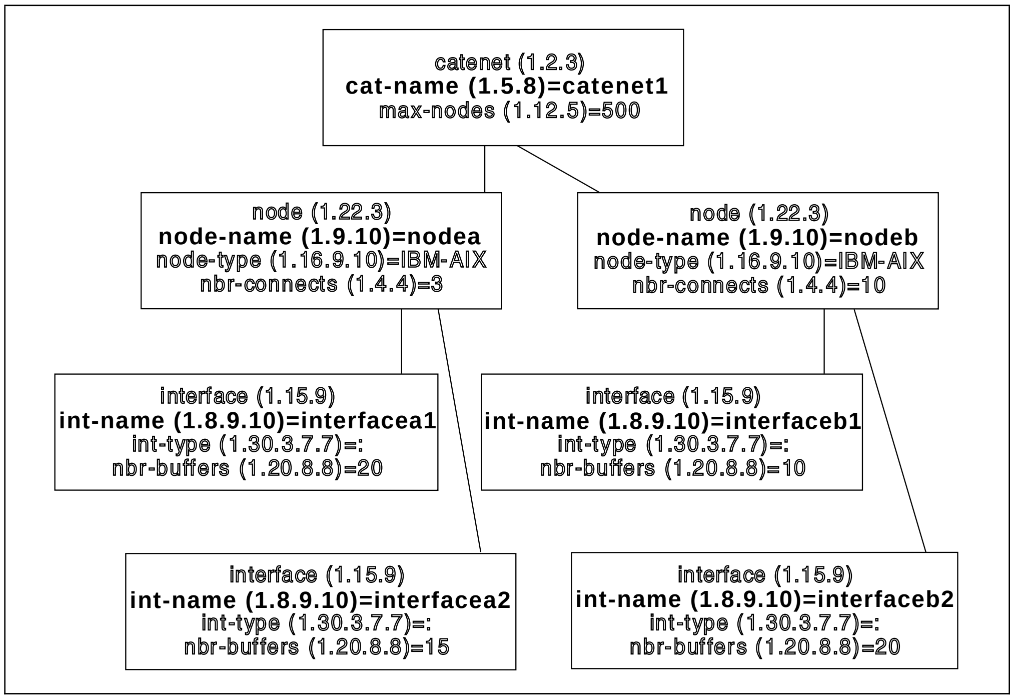

Figure 9. A Containment (Naming) Tree

For example, in Figure 9, consider the object instance for the computer workstation named nodeb. The object instance is defined based on the computer workstation object class, which has the example registration ID 1.22.3.

The object instance is part of a larger environment. The workstation contains two interface cards (interfaceb1 and interfaceb2), and it is contained within a LAN named catenet1.

The containment relationship is the hierarchy by which object instances are contained in others. The containment relationship implies that an object exists only as long as its superior object exists. Deleting a managed object can cause the implicit destruction of all its subordinate objects.

Every object class has a distinguishing attribute that is used to uniquely identify each instance of the class. An object instance's distinguishing attribute, together with the value of that attribute, is called the *relative distinguished name* (RDN) of the object instance.

By traversing the containment tree, also called the naming tree, from the root to an object instance, you can construct a sequence of RDNs that is unique to the object instance. This unique sequence is called the *distinguished name* (DN) of the object instance. The DN is also sometimes referred to as the *fully distinguished name* (FDN).

Again, see Figure 9 on page 182:

- *int-name* is the distinguishing attribute of the interface object class. Its example registration ID is 1.8.9.10.
- *node-name* is the distinguishing attribute of the node object class. Its example registration ID is 1.9.10.
- *cat-name* is the distinguishing attribute of the catenet object class. Its example registration ID is 1.5.8.

The RDN of the interface card named interfacea2 is:

```
{ 1.8.9.10=interfacea2 }
```

Assuming that this catenet object instance is an immediate subordinate of the root, the FDN of this interface card is:

```
{ 1.5.8=catenet1, 1.9.10=nodea, 1.8.9.10=interfacea2 }
```

The RDN of an object instance is unique only under its superior; the FDN of an object instance is globally unique.

Note: The containment tree is also referred to as the naming tree since it is used to uniquely name object instances. Throughout the remainder of this book, the term *naming tree* is used.

The Inheritance Relationship

The inheritance relationship describes which object classes are derived from others. When one object class is derived from another, it inherits the characteristics of that parent class. The inheritance relationship is used to define new object classes based on existing ones. The inheritance relationship is typically applied to OSI object definitions, not to IP or SNMP objects.

Object classes are arranged in an hierarchy. A class that is the ancestor of another class is called a *superclass*. A class that is a child of another class is called a *subclass*. Every subclass inherits all the characteristics of its immediate superclass and all superclasses higher up in the inheritance hierarchy. Figure 10 on page 184 shows a sample inheritance tree.

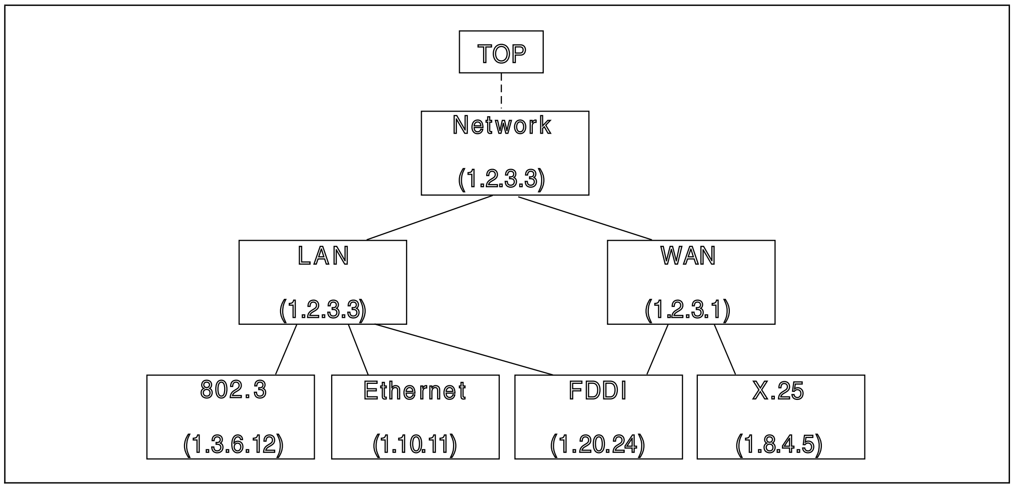

Figure 10. An Inheritance Tree

There may be cases where a new object class requires characteristics present in existing object classes that are not hierarchically related. To address this, an object class is allowed to have more than one superclass. When an object inherits characteristics from more than one class, it inherits the merged set of characteristics. This is called *multiple inheritance*.

In the inheritance tree diagram in Figure 10, the FDDI object class needs characteristics that are present in both the LAN and WAN object classes. The FDDI object class is constructed using multiple inheritance. The FDDI object class contains the merged set of characteristics from both the LAN and WAN classes.

The root node in the inheritance tree is treated specially. It is the ancestor to all other object classes and has no superclass. In the diagram, the root object class is not shown. The Network object class is a descendant of the root object class.

Relationship Summary

The following list summarizes the relationships and their purpose:

Registration Registration applies to the definition of object classes. The class of an object instance is specified by a registration ID number, assigned by a registration authority. As you traverse the registration hierarchy from the top to a particular object class, the sequence of registration ID numbers along the path forms an identifier that is unique for the object class.

Naming The naming relationship is also called the *containment relationship*. Managed objects can contain other managed objects. This hierarchy of containment is used to derive a unique name for each object instance. Each object has a relative distinguished name (RDN), which might not be globally unique.

However, the concatenation of RDNs in the chain of containment forms a unique name, the distinguished name (DN), which is globally unique.

For example, if object A contains B, which contains C, which contains D, the distinguished name of D is *A.B.C.D*; no other object instance can have that name.

Inheritance	Managed-object classes can derive their attributes, actions, and notifications by inheriting the definitions of attributes, actions, and notifications from other objects. The inheritance relationship defines the hierarchy of derivation for object classes. The resulting tree is similar to a genealogy of object classes.

Scoping and Filtering Requests

The CMIS services allow you to target a request to a specific object or group of objects in the naming tree. The targeting process involves two mechanisms: scoping and filtering.

Scoping is used to identify the target object or objects for a request, based on their containment relationship to a reference object instance. *Filtering* applies logical tests to further refine the list of candidate objects. Although scoping and filtering are frequently discussed together, they are independent.

Scoping

Scoping is available for the following CMIS operations:

- Action request
- Delete request
- Get request
- Set request

When you use scoping, the base object in the naming tree is a reference point for the scoping. Then you apply one of the five scoping levels:

Base object	This is the default scope, used whenever the scope is unspecified.
First level only	This selects all the immediate subordinates of the base object.
Whole subtree	This selects the base object and all of its subordinates.
Individual level	This selects all the *n*th level subordinates of the base object. The immediate subordinates are first level; their immediate subordinates are second level; and so on.
Base to *n*th level	This selects the base object and all of its subordinates through the *n*th level.

Filtering

A filter consists of logical tests that are applied to each member of the list of object candidates. Filtering extracts a subset of the managed objects from the list generated during scoping.

Agent Requirements under OSI

The following minimum requirements of an agent help ensure interoperability and are all related to participation in the OSI management environment. The agent must:

- Support all the operations defined for the managed object
- Detect and report all errors that can occur in the object
- Recognize scoping and filtering and return error codes if scoping and filtering are not supported

Handling Errors

Handling errors is a major responsibility for agents. There are many different CMIS and SNMP errors that an agent needs to recognize and return in order to be interoperable. These are defined in the XMP header files as constants prefixed with `MP_E_`.

The real challenge in handling errors is to map the errors you detect to the CMIS errors. Sometimes there is an obvious correlation and other times it is not so obvious. You need to correlate error conditions to CMIS errors as thoroughly as possible.

If your agent encounters an error that cannot be mapped to one of the defined errors, but must be returned, return the MP_E_PROCESSING_FAILURE error.

Another type of error that can occur is the failure to successfully get or set part of a list of attributes. In this case, you need to return the list with correct values for those attributes that the operation could successfully get or set, with error indicators for the rest. These errors are returned using the CMIS-Get-List-Error and CMIS-Set-List-Error functions.

Refer to the *NetView for AIX Programmer's Reference* for a description of when to generate each of the CMIS errors.

Handling Scoping and Filtering

Scoping and filtering of requests are described in “Scoping and Filtering Requests” on page 185. If your agent does not support scoping or filtering, you must at least return an error when a request includes these features.

If your agent does support either of these features, there are minimum requirements for each of them to ensure interoperability.

Scoping: Supporting scoping can be a complex task. Scoped requests are aimed at either the base object or subordinates contained in it. For example, if an agent for a workstation receives a scoped request, it is for something contained in the workstation, such as a LAN Interface Card (LANIC) object. If your agent manages both the workstation and the LANIC objects, it can complete the operation.

However, if the agent manages only the workstation, it must forward the request to the agent responsible for the LANIC objects contained in the workstation. This forwarding is done by issuing one or more appropriately rescoped requests. Finally, when all responses to these requests have returned, the agent must forward the list of the responses back to the requesting management program.

The higher an object is in the naming tree the more difficult it is to support scoping, because all subordinate objects must be considered. If your agent supports scoping, then all of the scoping options should be valid; scoping should not be partially implemented.

Filtering: Supporting filtering can also be a complex task. There are three choices:

- You can choose not to support filtering at all and return an error when you receive a request with a filter specified.

- You can choose to meet the following minimum requirements for supporting filtering:
  - Support *equality*, *greater-or-equal*, *less-or-equal*, and *present* as the test conditions when specified by the object class definitions.
  - Support the AND and OR operators for a pair of conditions.
- You can choose to support filtering beyond the minimum requirements. For example, you could include support of substring matching, set comparisons, NOT operators, and nested ANDs and ORs. Remember to keep the design of your agent focused on supporting the object definition.

Refer to the *ISO IS 9595, Common Management Information—Service Definition* for more information about filters and filtering.

Synchronization

When a manager (or object manager) uses scoped get or set requests, the operation might fail on one or more of the selected objects. Thus, in addition to the scope, the requester must also specify how the responder should deal with this condition. The two possible ways to handle partial failure are:

Atomic retrieval	Under this scheme, all retrievals are checked to determine whether they succeed. If any of them fail, no retrievals are performed. Otherwise, all retrievals are performed.
Best-effort retrieval	Under this scheme, all retrievals are attempted. Those that fail return an error; those that succeed are returned.

This general principle is called *synchronization*. Your agent must recognize the type of synchronization requested and respond accordingly. For example, consider an object manager that receives a scoped request with atomic synchronization. It then issues several appropriately re-scoped requests. If any of these fail, the object manager cancels any outstanding requests, issues a service error, and sends a message to the requester to terminate the request.

Developing Network Management Applications

The NetView for AIX program provides a rich set of tools for your use in developing network management applications. The remaining chapters in this section describe these components:

Chapter 12, “Using the XOM API” on page 189 describes the use of the XOM API to manipulate data structures for XMP.

Chapter 13, “Using the XMP API” on page 213 describes the use of the XMP API to create network management programs.

Chapter 14, “C-Language Binding for the XOM and XMP APIs” on page 231 describes the C-language binding to the XOM and XMP APIs.

Chapter 17, “Using the NetView for AIX SNMP API” on page 281 and Chapter 18, “Using SNMP API Functions and Data Structures” on page 289 describe the use of the NetView for AIX SNMP API.

Chapter 19, “Filtering Network Events” on page 307 describes the use of the NetView for AIX Filter File API and filter registration routines to register your application to receive events using filters.

Chapter 20, “Using the General Topology Manager” on page 315 describes the use of the NetView for AIX Topology MIB to monitor networks that do not use the IP protocol.

Chapter 21, “Communicating with the General Topology Manager” on page 343 describes the use of the trap interface and the GTM API to communicate with the General Topology Manager.

Chapter 12. Using the XOM API

This chapter describes the OSI-Abstract-Data Manipulation application programming interface (the XOM API). If you are new to XMP and XOM, you are encouraged to read this chapter from beginning to end. These APIs are not especially difficult to use after you have mastered them. However, these interfaces, especially XOM, introduce terms, concepts, and techniques that may seem strange at first. By skipping forward, you might become confused.

Understanding XOM

XMP employs XOM to create, examine, modify, and delete the arguments to XMP functions. This section introduces you to the XOM API.

XOM provides a generalized data-handling mechanism. It was designed for use in conjunction with application-specific OSI APIs, such as XMP. Most of the data types manipulated by the XOM API arise from ASN.1 definitions, but some are convenience objects.

ASN.1 is the OSI-defined standard for describing a set of named and defined data types that can be as simple as integers, characters, and Booleans or as complex as arrays and structures. ASN.1 includes ways to create new types based on the predefined types. Basic encoding rules (BER) define how ASN.1 types are encoded to be transported over the network.

Benefits of Using XOM

Direct manipulation of ASN.1 data can be time-consuming and prone to errors. One of XOM's main purposes is to hide as much of ASN.1's complexity as possible, including the BER.

One of the key benefits of XOM is its generality; after you have learned how to work with an interface that uses XOM, such as XMP, you can transfer that skill to any other OSI API that uses XOM (such as XDS, the X/Open programmatic interface to X.500 directory services).

Object-Oriented Terminology in XOM and XMP

XOM has some object-oriented characteristics. The OSI model of network management is also object-oriented. Because both XOM and XMP use object-oriented terminology, there is some potential for confusion.

This book adopts the lexicons (terms and function names) of the XOM and XMP specification books. However, the lexicon of XOM is very similar to that of XMP. For example, XOM defines an om_create() function, and XMP defines an mp_create_req() function; the two have very different usage and effect. This book differentiates between the XOM and XMP lexicons; occasional special notes in this chapter point out the distinction.

The Strategy and Structure of XOM

An implementation of XOM exists only as part of the implementation of another API; XOM cannot, by definition, exist independently of another API. An API, such as XMP, that uses XOM specifies two key components:

- A set of functions specific to the API. For XMP, this set includes the CMIS and SNMP service functions and several utility functions. These functions are described in "The Protocol Services of XMP" on page 214.
- A set of structured information objects that constitute the parameters and results of the API functions. These objects are defined in a package of class definitions. XOM packages are described in "XOM Packages" on page 192. The essential purpose of XOM is to manage these information objects.

XOM can relieve you, the programmer, of having to define and deal directly with large and complex data structures. The XOM functions provide you with tools to operate on substructures of the parameters and results of the XMP functions.

XOM and Object Orientation

This section describes the XOM information objects, called *OM objects*. There are a few unavoidable forward references to the XOM functions that manipulate OM objects. Later sections develop this material in more detail.

XOM is designed to provide another way to represent abstract data. Briefly, an OM object is nothing more than a modest, logical, C-language data structure used to express abstract data types that reflect the underlying data representation defined by ASN.1.

As a C-language programmer, you are accustomed to using arrays, structures, and unions to create complex data types. XOM defines a new set of tools to achieve this task, hiding the underlying ASN.1 implementation.

Carefully note the definitions in the following list:

OM attribute

OM attributes are the building blocks of OM objects; each OM attribute is one component of an OM object. For example, an OM attribute that stores an IP address could be one component of an OM object that represents a network address. An OM attribute is represented as a C language structure called an OM descriptor. It has three parts: (1) the type (name) of the OM attribute, (2) its syntax, and (3) its value. The syntax denotes the data type of the value (as an integer, Boolean, or string, for example).

To continue the example, an OM attribute intended to store an IP address might have the type *ip-Address*, the syntax OCTET STRING, and a value that is a string length of four, containing, in order, the four octets of a particular IP address.

OM object

An OM object consists of a list of OM attributes. It is represented in the C language as an array of OM descriptors. The OM class of the OM object specifies the number of OM attributes on the list and the type and syntax of each.

Each attribute of a managed object is typically represented as an OM object. (The distinction between OM objects and managed objects is crucial here.) Many OM attributes are themselves OM objects; these are called subobjects.

OM class An OM class defines the list of OM attributes that make up a particular kind of OM object. As noted before, the OM class of the OM object specifies the number of OM attributes in the list and the type and syntax of each.

You can think of OM classes as the data types for OM objects; by this model, any given OM object is a variable whose data type is specified by its OM class. Do not forget, however, that an OM object is just a list of OM attributes.

Many of the parameters and results of XMP and XOM functions are OM objects, which by definition are members of some OM class.

Note: Do not confuse the terms in the preceding list with the object-oriented terms from the domain of network management. For example, note that an OM object is a data structure, but a managed object is a resource to be managed. The distinction between these two sets of terms must be remembered to avoid confusion.

Throughout this book, care has been taken to avoid such confusion. The terms *OM object*, *OM class*, and *OM attribute* always carry the OM prefix. Whenever the terms *object*, *class*, or *attribute* appear without the OM prefix, they refer to the network management domain, not to XOM.

Inheritance in XOM

Because XOM is object-oriented, it includes the notion of inheritance. OM classes are related in a hierarchy of superclass, class, and subclass; this hierarchy forms a strict tree structure, without cross-links. See Figure 11 on page 192.

A *subclass* of an OM class inherits all of the OM attributes of the parent OM class. The parent OM class is called the *superclass* of the subclass.

There is an important relationship between an OM class and its superclasses. An instance of any OM class is always considered to be an instance of each of its superclasses. This means that XOM and XMP functions can accept any subclass of the OM class of an input parameter. Likewise, these functions can return any subclass of the OM class of an output parameter. XOM provides a special function, om_instance(), to determine exactly to which OM class an OM object belongs. For information about this function, refer to the *NetView for AIX Programmer's Reference*.

For example, XMP defines a function called mp_get_req(), described in “The Protocol Services of XMP” on page 214. The mp_get_req() function works for both CMIS and SNMP get operations. To accommodate this, there is a Get-Argument OM class with two subclasses: CMIS-Get-Argument and SNMP-Get-Argument. The mp_get_req() function requires a Get-Argument and accepts either a CMIS-Get-Argument or a SNMP-Get-Argument. This is how XMP can serve both CMIS and SNMP users.

Some OM classes in the OM class hierarchy are abstract, while others are concrete. Abstract OM classes exist as a means to provide, through inheritance, common OM attributes to a group of concrete classes. You cannot create an OM object whose OM class is abstract; you can create only an OM object whose OM class is concrete. In the previous example of the two types of get arguments, the Get-Argument OM class is abstract. Only its subclasses, CMIS-Get-Argument and SNMP-Get-Argument, are concrete. In Figure 11, both the abstract and concrete OM classes have the prefix MP_C_ and are fully capitalized. This reflects the form of the C-language identifiers used for abstract and concrete OM classes.

The root of the OM class hierarchy is the abstract OM class Object, the only OM class that has no superclass. All other OM classes are subclasses of the OM class Object.

The only OM attribute of the abstract OM class Object is the class, which consequently is inherited by all OM classes. In every OM object, this attribute identifies the class of the OM object. This attribute appears in the C-language structure for every OM object.

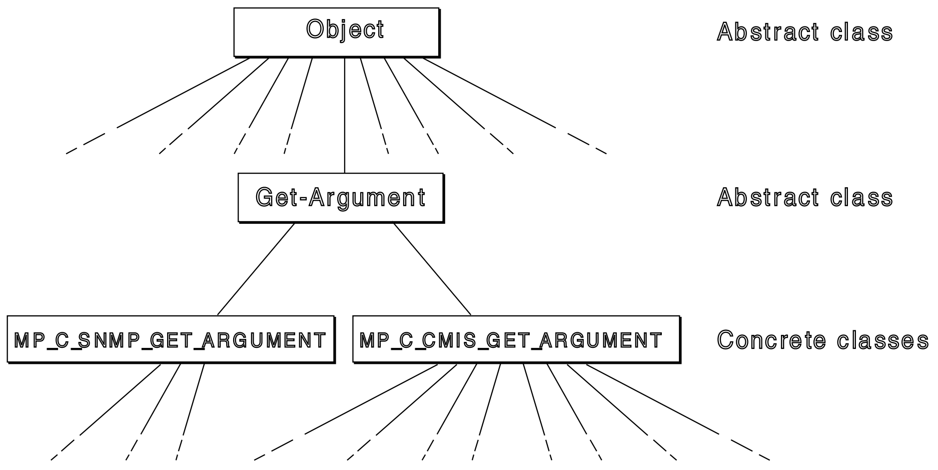

Figure 11. Inheritance and Class Types in XOM

XOM Packages

An XOM package is a functionally related set of OM classes. In the XMP interface, there are four base packages:

- The OM package
- The Common package
- The CMIS package
- The SNMP package

OM class definitions in the OM and Common packages are automatically included in both the CMIS and SNMP packages, as explained in “About XOM Package Closures” on page 193.

The class definitions for these packages appear in the *NetView for AIX Programmer's Reference*. Each class definition consists of the following:

- A description of the OM class, which identifies its superclasses, and whether the OM class is abstract or concrete.

- A table that lists each OM attribute that is specific to the OM class. Each table entry includes:
  - The name of the OM attribute
  - The syntax of the OM attribute
  - Any constraint on the length (applicable only to string-type syntaxes)
  - Any constraint on the possible number of values for the OM attribute:
    - An optional OM attribute has either *0-1* or *0 or more* as a constraint.
    - A mandatory OM attribute has either *1* or *1 or more* as a constraint.
- An elaboration of the purpose, use, and potential values for each OM attribute.
- A list of which attributes are mutually exclusive.

"Declaring OM Objects" on page 206 explains in more detail how to use the class definitions in the *NetView for AIX Programmer's Reference*.

About XOM Package Closures

The term *package closure* appears occasionally in the literature about XOM. The *closure* of a package (P) is the set of all packages that the package (P) requires in order to resolve all its OM class definitions.

For example, each OM class defined in the CMIS package includes, as its first attribute, the object identifier. This attribute relates to the OM class Object, which, as noted before, is the root of the OM class hierarchy. This class is defined in the OM package. Therefore, the closure of the CMIS package, or any other package, must include the OM package.

It is through the notion of a package closure that the CMIS package automatically includes the Common package and the OM package. This is illustrated in Figure 12.

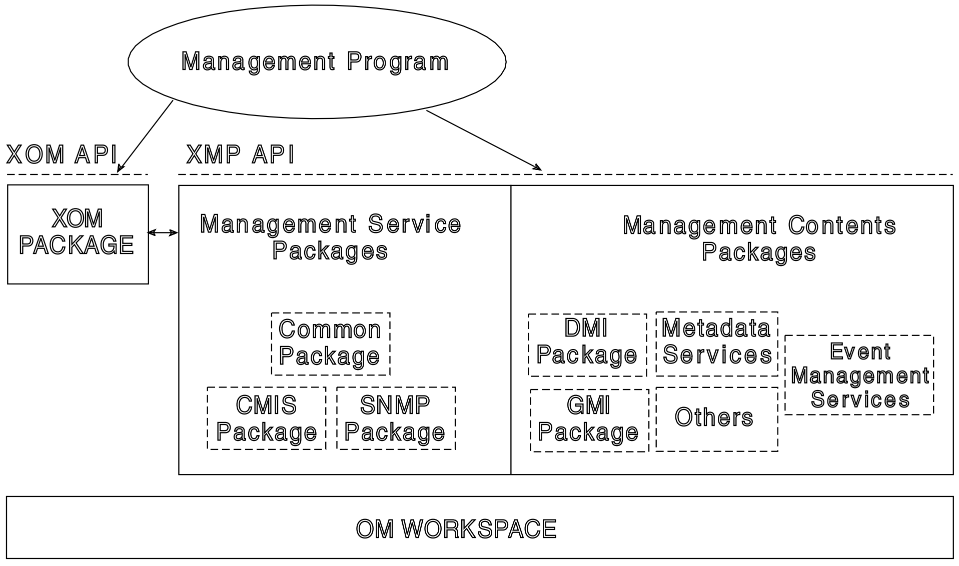

Figure 12. Package Closures for CMIS and SNMP Packages

Using OM Packages in Your Management Program

When your management program uses the XMP API, the Common and OM packages are automatically available. You can include other packages in your management program. This requires two steps:

1. Provide the package name to your management program with the OM_EXPORT macro, which is defined by XOM. For example:

   ```
   OM_EXPORT (MP_SNMP_PACKAGE)
   OM_EXPORT (MP_CMIS_PACKAGE)
   ```

2. Bind the package to your workspace by calling the mp_version() routine. See "XMP Workspaces" on page 224 for more information on this routine.

For more information about OM_EXPORT and other XOM macros, see "XOM Object-Definition Macros" on page 203. For more information about the `mp_version()` function and other XMP functions, see "The Protocol Services of XMP" on page 214 and "The Supporting Functions of XMP" on page 215. The OM_EXPORT macro and mp_version() routine create a series of declarations and actions that give you access to the class definitions in the CMIS, SNMP, and the Management Contents Packages, which are defined in the *NetView for AIX Programmer's Reference*. These two steps give you access to the OM management services packages through the XMP API. OM_EXPORT and mp_version() can also be used to access the NetView for AIX management contents packages through the XMP API.

The Management Contents Packages

The management contents packages contain object classes that are related to the specific management information handled by agents and managers. These packages describe the mapping of the attribute values to XOM classes.

The OM classes in a Management Contents Package are just like any other OM class. The way you use an OM class is not affected by the fact that it does not belong to the base XMP packages (OM, COMMON, CMIS and SNMP).

The packages provided in this implementation are listed below, and are described in the *NetView for AIX Programmer's Reference*.

- *ISO 10165-2, Management Information Services—Structure of Management Information Part 2: Definition of Management Information*
- LNV package - IBM LAN NetView - definitions of attributes and OM classes for interoperability with HP agents

XOM Data Objects

XOM represents the data it manipulates as OM objects. As the developer of a network management program, you need to be able to create and discard these OM objects, and to examine and exchange data with them, in whole or in part.

Public and Private OM Objects

XOM defines two types of OM objects, public and private, which are described in the following list:

Public OM objects
: A public OM object consists of a special C-language data structure that is directly accessible by a management program. A public object can be defined by a management program or by XOM. XOM public objects exist in a workspace that is accessible by a management program. Values of OM attributes in public OM objects can be directly inspected by the management program. If an attribute belongs to a public object defined by the program, the program can use a regular assignment statement to change the attribute's value.

  The output of an XOM or XMP function is not a public OM object, except for the result of an om_get() function, which is a public OM object.

  Public OM objects can be translated into private OM objects with the om_put() routine.

Private OM objects
: A private OM object is maintained in an XOM workspace, and its representation is hidden from a management program. Private objects are supplied because they are easier to use by XMP functions. The XOM API functions give you indirect access to a private OM object; you cannot manipulate it directly. For example, to change the value of an OM attribute in a private OM object, you would use the XOM om_put() function, rather than an assignment statement.

  Private OM objects can be translated into public OM objects with the om_get() routine.

OM objects that are passed to and from XOM and XMP functions are allowed to be private. Most functions also allow these OM objects to be public. Sometimes they can be constants, such as MP_DEFAULT_SESSION. Any OM object that is the result of XOM and XMP functions, with the exception of om_get(), is private. You can create an equivalent public object with a call to om_get().

Both public and private OM objects are accessed through a C-language pointer. In the case of a private object, the pointer is used only as a function parameter; for a public object, the pointer refers to the data structure containing the OM object.

Mixed Public and Private OM Objects

OM objects are frequently complex, consisting of several other OM objects, which can consist of still other OM objects. It is possible to have an object that mixes public and private elements. For example, as illustrated in Figure 13 on page 196, the mp_action_req() function uses an OM object parameter of OM class MP_C_CMIS_ACTION_ARGUMENT. One field of this OM object is another OM object of OM class MP_C_ACTION_INFO. This OM object has a further OM object field of OM class MP_C_ACTION_TYPE_ID.

Even if the MP_C_CMIS_ACTION_ARGUMENT OM class is public, any of its OM object members, such as the MP_C_ACTION_INFO field, can be private. However, the reverse (MP_C_CMIS_ACTION_ARGUMENT, private;

MP_C_ACTION_INFO, public) cannot be true; no component of a private OM object can be public.

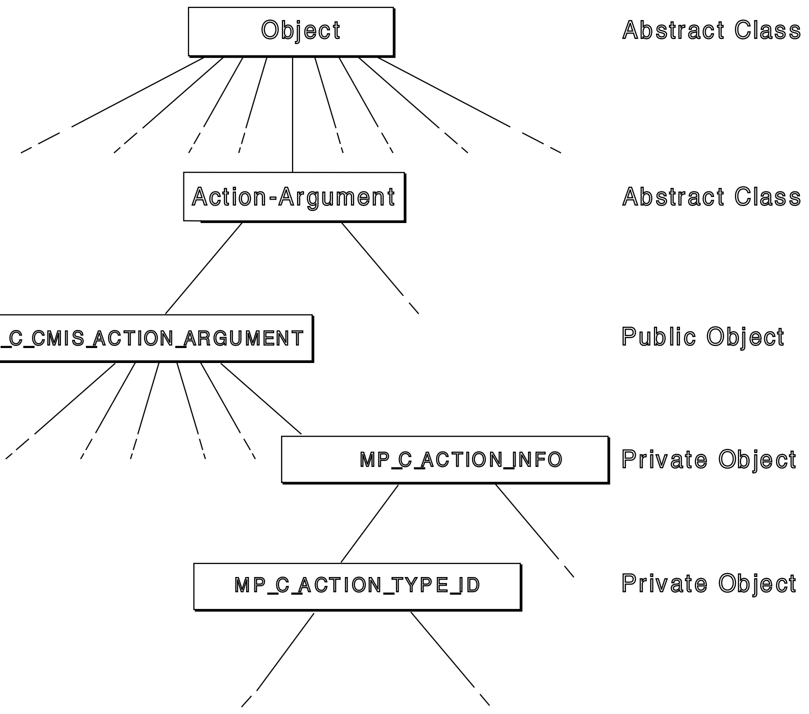

Figure 13. Mixed Public and Private OM Objects

Types of Public OM Objects

There are two types of public OM objects. They are defined in the following list:

Service-generated
A service-generated public OM object should be considered read-only. It is public, but it exists in the workspace that XOM provides. Service-generated public OM objects are created when you request a public copy of some private OM object. This is a common action, because although OM objects that are the output of XMP and XOM functions are private, your management program needs to work with a public OM object. This is because the structure of private OM objects is hidden from the application, so that only public OM objects can be inspected by your program.

Client-generated
A client-generated public OM object exists in a workspace that you provide through memory allocation or static declaration.

The key difference between these two types of public OM objects is the way you treat the memory involved.

Memory Management for OM Objects

OM objects are accessed through a C-language pointer. How the memory addressed by this pointer must be treated depends on whether it is controlled by XOM or by your management program. The following list indicates how memory should be managed for each type of OM object:

Client-generated public OM objects
You have full responsibility for memory management for this type of OM object. If you allocate memory, you must later deallocate it. Failure to do so results in memory leaks, which can result in management program failures caused by resource consumption.

Service-generated public OM objects
XOM has full responsibility for memory management for this type of OM object. XOM allocates memory when you create the public version of a private OM object with the om_get() routine, and deallocates it when you delete the public OM object with the om_delete() routine. Treat this type of OM object as read-only.

Destroying the XOM workspace through mp_shutdown() does not affect service-generated public OM objects; you must recover the memory associated with each OM object by calling om_delete() for each object.

Private OM objects
XOM has full responsibility for memory management for this type of OM object. Whenever you create new private OM objects, you should delete them when they are no longer needed.

The function call mp_shutdown() automatically deletes all private objects associated with the workspace. Destroying the XOM workspace with mp_shutdown() does recover the resources associated with private objects.

Warning Do not modify the pointer to a private OM object or anything to which it points. Doing so can cause unpredictable, catastrophic errors.

Also, the data in a service-generated public OM object must be considered read-only. Do not directly modify the pointer to a service-generated public OM object, or anything to which it points. Modifying either the pointer or the OM object to which it points can cause unpredictable, catastrophic errors.

There is one exception: if the service-generated public OM object has, as one of its elements, a private OM object, you can use the private subobject as an argument to an XOM function, such as om_put(), that modifies it.

Using Public and Private OM Objects

No management program can deal exclusively with either public or private OM objects; you must use both types in every case. However, you can focus your program on one approach or the other. Your decision depends on the type of program you are writing.

When to Use Public OM Objects

Public OM objects let you use static data structures, instead of making multiple calls to XOM functions to construct private OM objects dynamically. Therefore, public OM objects are, in general, used by management programs filling a responder role, because the requirements of such a program are generally well defined.

When to Use Private OM Objects

Private OM objects let you dynamically create data structures by calling XOM functions, instead of having to define static data structures to cover all possibilities. Therefore, private OM objects are, in general, appropriate for management programs filling a requester role. Since the requests issued by these programs are often based on the actions of a human user, it is impossible to know in advance what structures might be needed.

Manipulating Objects with XOM

The XOM API implements the XOM concepts described in this chapter. This section introduces the programmatic interface to XOM, including the functions and key data structures. Many of these data structures are defined in the following Management Services Packages header files:

- *xom.h*
- *xmp.h*
- *xmp_cmis.h*
- *xmp_snmp.h*

When using APIs like XOM and XMP, you should know the conventions used to assign C-language identifiers. Table 14 shows the conventions used by XOM in assigning C-language identifiers. XMP uses similar conventions, which are noted in Table 18 on page 213.

Table 14. XOM Interface Naming Conventions

Item	Convention	Example
Functions	The names of XOM functions are prefixed with `om_` and are in lower-case letters. In this book, they are further distinguished by a trailing pair of empty parentheses.	`om_get()`
Function parameters and results	The names of XOM function parameters and results are in lower-case letters, italicized, and have no special prefix.	*string_offset*
Data types	The names of XOM data types are prefixed with OM_ and are in lower-case letters.	OM_return_code
Macros and defined constants	The names of XOM macros and defined constants are prefixed with OM_ and are in upper-case letters.	OM_OID_DESC
Syntaxes	The names of XOM syntaxes are described in “Attribute Syntax” on page 200. They are prefixed with OM_S_ and are in upper case letters.	OM_S_ENUMERATION

Note: Whenever this book states the prefix for a set of identifiers, the entire prefix is stated. For example, suppose a set of identifiers carries the prefix OM_. Identifiers with the prefix OM_S_ would not be members of the designated set.

OM Object Descriptors

Each OM class consists of a list of OM attributes. Each OM attribute is in turn represented as a C-language data structure, called the OM descriptor, whose declaration is as follows:

```
typedef struct OM_descriptor_struct
{
  OM_type     type;
  OM_syntax   syntax;
  OM_value    value;
} OM_descriptor;
```

Therefore, an OM object is an array of OM descriptors. Figure 14 illustrates the explanation that follows.

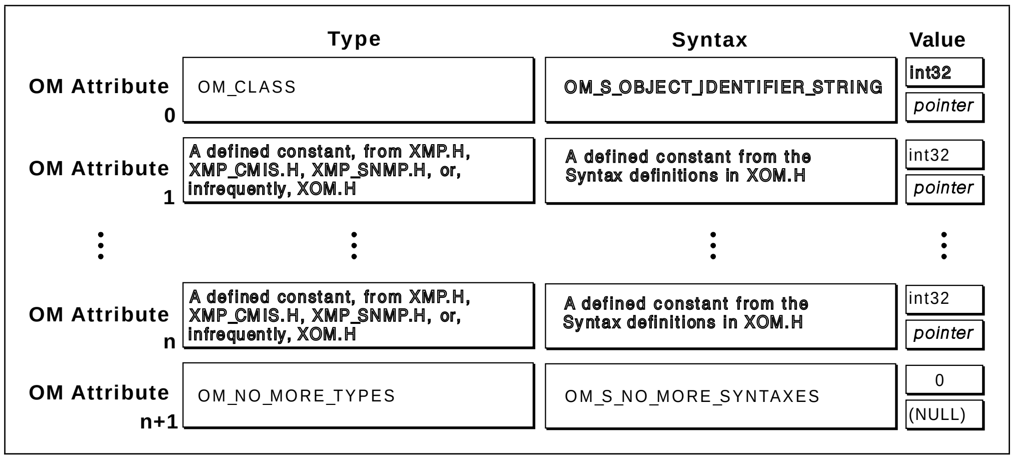
Type
Syntax
Value
OM Attribute 0
OM_CLASS
OM_S_OBJECT_IDENTIFIER_STRING
int32
pointer
OM Attribute 1
A defined constant, from XMP.H, XMP_CMIS.H, XMP_SNMP.H, or, infrequently, XOM.H
A defined constant from the Syntax definitions in XOM.H
int32
pointer
OM Attribute n
A defined constant, from XMP.H, XMP_CMIS.H, XMP_SNMP.H, or, infrequently, XOM.H
A defined constant from the Syntax definitions in XOM.H
int32
pointer
OM Attribute n+1
OM_NO_MORE_TYPES
OM_S_NO_MORE_SYNTAXES
0
(NULL)

Figure 14. Representation of an OM Object

An OM descriptor contains three elements, which are described in the following list:

type The type specifies the name of the OM attribute. This field always contains a defined constant, which can be found in the header files related to the package containing the OM attribute. The OM attributes in *xom.h* are prefixed with OM_, and those in *xmp.h, xmp_cmis.h*, and *xmp_snmp.h* are prefixed with MP_. The OM attributes in the Management Contents Packages will have different prefixes, such as DMI_ for the DMI package.

syntax The syntax specifies how the value in the value field should be interpreted according to the ASN.1 syntaxes supported by the XOM specification. It always contains a defined constant, one of the 21 OM syntaxes specified in *xom.h*. The syntax always begins with the prefix OM_S_.

value This field contains the attribute value. It is a union containing a 32-bit value that can have several interpretations, as specified by the syntax, and a pointer to a string or object. Readers familiar with ASN.1 should note that this field contains both the length and value.

If the syntax field specifies any syntax that is not a string-type or object, the pointer field is NULL, and the 32-bit field contains the value of the attribute. This value may be an integer, a Boolean value, or whatever else the syntax field specifies.

If the syntax field specifies a syntax that is a string-type, the pointer field points to the beginning of the string, and the 32-bit field is an integer that specifies the length of the string (number of bits in a bit-string, or octets in an octet- or character-string). If the syntax is defined as object, the 32-bit field is zero and the pointer points to an OM object.

The first and last OM attributes of an OM object have special significance:

- The first OM attribute , shown as OM Attribute$_0$ in Figure 14 on page 199, is inherited from the root OM object, OM_C_OBJECT. This attribute identifies the class of the object. The type and syntax fields are always the same, OM_CLASS and OM_S_OBJECT_IDENTIFIER_STRING. The value field varies, depending on the type of the OM object. The names of the OM classes are also found in the header files (*xmp.h*, *xmp_cmis.h*, *xmp_snmp.h*, *omp_dmi.h*, and *lnv.h*), and their prefixes also obey the same rules defined in the two previous lists.
- The last OM attribute , shown as OM Attribute$_{n+1}$ in the previous figure, is always a null descriptor used to terminate the attribute list for the OM object.

Attribute Syntax

The previous section showed that the second field of an OM descriptor specifies the syntax of the attribute. The XOM specification defines the important syntaxes described in Table 15.

Table 15 (Page 1 of 2). Key XOM Syntaxes

Syntax Category	**Defined Values**	**Description**
Boolean	OM_S_BOOLEAN	The value for an attribute of this syntax can be either *OM_FALSE* or *OM_TRUE*.
Enumeration	OM_S_ENUMERATION	The value for an attribute of an enumerated syntax is one of a set of distinct values defined for the particular attribute.
Integer	OM_S_INTEGER	The value for an attribute of this syntax is an integer.
Null	OM_S_NULL	This is a no-value placeholder.
Object	OM_S_OBJECT	The value for an attribute of an object syntax is an OM object. It can be any instance of a class associated with the syntax.

Table 15 (Page 2 of 2). Key XOM Syntaxes

Syntax Category	Defined Values	Description
String	OM_S_BIT_STRING OM_S_ENCODING_STRING OM_S_GENERAL_STRING OM_S_GENERALISED_TIME_STRING OM_S_GRAPHIC_STRING OM_S_IA5_STRING OM_S_NUMERIC_STRING OM_S_OBJECT_DESCRIPTOR_STRING OM_S_OBJECT_IDENTIFIER_STRING OM_S_OCTET_STRING OM_S_PRINTABLE_STRING OM_S_TELETEX_STRING OM_S_UTC_TIME_STRING OM_S_VIDEOTEX_STRING OM_S_VISIBLE_STRING	A string is an ordered sequence of zero or more bits, octets, or characters. The value for an attribute of any syntax in this category is a string whose form and meaning are associated with the particular syntax. The length of a string depends on the string type. It is the number of bits in a bit string, or the number of octets in an octet or character string. Multi-octet characters are possible. String length is confined to the interval {0, 2^{32}}. Any further constraints on the value length of a particular string are specified in the corresponding class definition.
Unlimited Integer	OM_S_UNLIMITED_INTEGER	The value for an attribute of this syntax is an integer that may be any size.
Real	OM_S_REAL	The value for an attribute of this syntax is a real number, composed of a positive or negative mantissa and an integer exponent.

Because it is not necessary to use all the bits to indicate the possible OM syntaxes, some bits in the syntax field can be used for other purposes. For this reason, the value in the syntax should be ANDed with the mask OM_S_SYNTAX before it is compared against any of the OM syntaxes listed above. The other bits are:

OM_S_LONG_STRING
: Significant only when the syntax is one of the String syntaxes. This bit indicates that the length of the string exceeds the local limit and should be manipulated by means of om_read/om_write.

OM_S_NO_VALUE
: Indicates that the value is not present, because an exclusion was requested in the om_get call.

OM_S_LOCAL_STRING
: Significant only when the syntax is String(*). This bit indicates that the string is represented in a local form.

OM_S_SERVICE_GENERATED
: Indicates that the object is in public format and was generated by XOM or XMP.

OM_S_PRIVATE
: Indicates that the object is in private format.

Attribute Value

The following data structure is used to store the value of an attribute:

```
typedef union OM_value_union
{
  OM_string      string;
  OM_boolean     boolean;
  OM_enumeration enumeration;
  OM_integer     integer;
  OM_real        real;
  OM_padded_object object;
} OM_value;
```

A data value of this data type is an attribute value. It has no components if the value's syntax is *no-more-syntaxes*, or if the value's syntax is *no-value*. Otherwise it has one of the following components:

String The attribute's syntax is a String syntax. The *OM_string* data type is defined as follows:

```
typedef struct
{
  OM_string_length  length;
  void              *elements;
} OM_string;

#define OM_STRING(string)
        {(OM_string_length)(sizeof(string-1),string)}
```

Length The number of octets by means of which the string is represented.

Elements The string's elements (the octets that make up its value).

In the C-language interface, the bits of a bit string are represented as a sequence of octets. The first octet stores the number of unused bits in the last octet. The bits in the bit string, commencing with the first bit and proceeding to the trailing bit, shall be placed in bits 7 to 0 of the second octet, followed by bits 7 to 0 of the third octet, followed by bits 7 to 0 of each subsequent octet, followed by as many bits as are needed of the final octet, beginning with bit 7.

```
                                  2nd octet        3rd octet

position of bit string:  0  1  2  3  4  5  6  7   8  9 ...

bit position in octet:   7  6  5  4  3  2  1  0   7  6 ...
                         ↑                    ↑
                        most                least
                     significant         significant
                         bit                 bit
```

In the C-language interface, a macro, *{OM_STRING}*, is provided for fabricating a data value of this data type, given only the value of its Elements component. The macro, however, applies to octet strings and character strings, but not to bit strings.

Boolean The attribute's syntax is OM_S_BOOLEAN.

False is denoted by zero *{OM_FALSE}*, true by any other integer, although the symbolic constant *{OM_TRUE}* specifically refers to the integer one.

Enumeration
The attribute's syntax is OM_S_ENUMERATION.

Integer The attribute's syntax is OM_S_INTEGER.

Unlimited Integer
The attribute's syntax is OM_S_UNLIMITED_INTEGER.

This attribute is represented as an OM_string data type (see description of OM_string above), where:

Length The number of octets by means of which the integer is represented.

Elements The octets which make up the value of the integer. These octets are encoded as if it were the Value field of an integer encoding according to Basic Encoding Rules (see ISO IS 8825).

Note: This new syntax was introduced to overcome a problem in the XOM specification which currently maps an Integer to a OM_uint32, limiting its value to the integers which can be represented in 32 bits. This is not a standard syntax and thus its use may cause problems or be difficult to port to other platforms. It is also subject to change as X/Open addresses this problem in future versions of XOM.

Real The attribute's syntax is OM_S_REAL.

Object The attribute's syntax is OM_S_OBJECT. The *OM_padded_object* data type is defined as follows:

```
typedef struct
{
  OM_uint32  padding;
  OM_object  object;
} OM_padded_object;
```

This data type uses its padding component to align the Object component with the Elements component of the String component. This facilitates initialization in the C language.

XOM Object-Definition Macros

An OM class name or package name is identified by an XOM object-identifier string. Details about these strings are presented in “Declaring OM Objects” on page 206. Like any other C-language variable, they must be defined in only one source file, and declared *external* in other source files that use them. There are two XOM object definition macros, OM_EXPORT and OM_IMPORT, that work together to simplify these declarations in your management program. Figure 15 on page 205 illustrates their relationship. Two other macros, OM_OID_DESC and OM_NULL_DESCRIPTOR, simplify the process of declaring an OM object.

The following list summarizes the XOM object-definition macros:

OM_EXPORT
The input parameter to this macro is a concrete OM class, or package name. Based on the contents of the header files, the macro generates all the associated declarations necessary to make the OM class, or package name available for your use.

For any given concrete OM class, or package name, this macro must appear in only one, compilation module of your management program. If you use it twice, the C-language compiler detects multiply defined symbols.

OM_IMPORT
Like OM_EXPORT, the input parameter to this macro is a concrete OM class, or package name. This name must appear as the input parameter for a call to OM_EXPORT in some other module. Based on the declarations established by that occurrence of OM_EXPORT, the OM_IMPORT macro generates all the associated external declarations necessary to match the declarations that OM_EXPORT established.

For any given OM class, or package name, this macro must appear in all modules that use that name except for the module where the name is a parameter for OM_EXPORT.

OM_OID_DESC
This macro declares the first attribute of an OM object, the Class attribute. It takes as parameters the type of an OM object, as described for OM descriptors, and the associated OM object-identifier string. It uses these parameters to declare an OM descriptor of the specified type and identifier.

The use of the macro OM_OID_DESC is not restricted to the first attribute of a class. It can be used to fill in the descriptor of any OM attribute whose syntax is Object Identifier.

OM_NULL_DESCRIPTOR
This macro is used to terminate the declaration of an OM object. It appears as the last OM descriptor of an OM object.

Using the OM_EXPORT and OM_IMPORT Macros

Figure 15 on page 205 illustrates the recommended way to use the OM_EXPORT and OM_IMPORT macros for exporting a name from one key source code (C-language) file and importing it in all others.

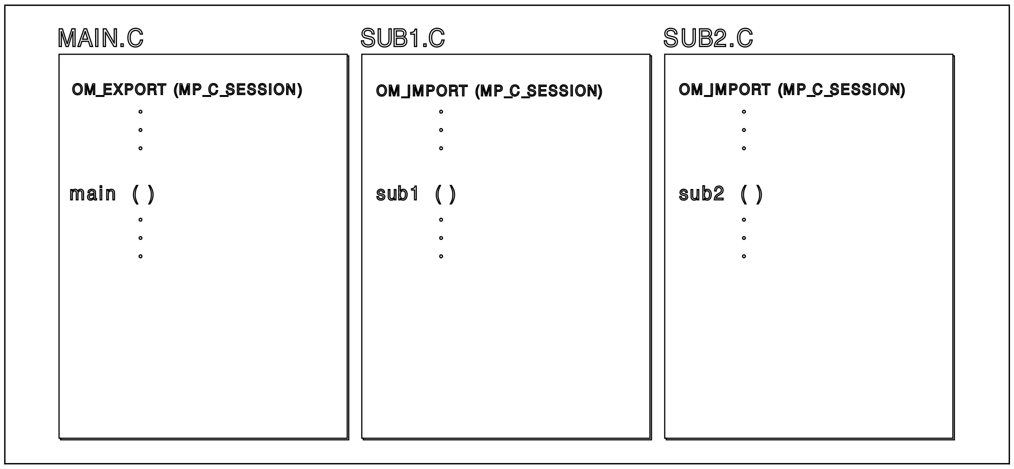

Figure 15. Using OM_EXPORT and OM_IMPORT

Remember these rules about OM_EXPORT and OM_IMPORT:

- The name of each concrete OM class, and package, that you use must appear in only one source code (C-language) file as a parameter to OM_EXPORT.
- The name of each concrete OM class and package that you use as a parameter to OM_EXPORT must appear as a parameter to OM_IMPORT in every other source code (C-language) file that uses the name.
- Both OM_EXPORT and OM_IMPORT must appear in the global declarations portion of the source code (C-language) file.

In most development environments, you can minimize confusion by designating one key module as the exporter; all other modules use OM_IMPORT exclusively.

Note: OM class identifiers appear in the header files with an additional prefix, OMP_O_. For example, the MP_C_SNMP_GET_ARGUMENT identifier appears as OMP_O_MP_C_SNMP_GET_ARGUMENT. This additional prefix is required by the OM_EXPORT and OM_IMPORT macros; you should not use it in your management program.

Using the OM_OID_DESC and OM_NULL_DESCRIPTOR Macros

The following code fragment declares an OM object, called action-error[], whose OM class is Action-Error:

```
static OM_descriptor action-error[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_ACTION_ERROR),
  {MP_MANAGED_OBJECT_CLASS, OM_S_OBJECT, {{0, NULL}}},
  {MP_MANAGED_OBJECT_INSTANCE, OM_S_OBJECT, {{0, NULL}}},
  {MP_CURRENT_TIME, OM_S_GENERALISED_TIME_STRING,  {0, NULL} },
  {MP_ACTION_ERROR_INFO, OM_S_OBJECT, {{0, NULL}}},
  OM_NULL_DESCRIPTOR,
};
```

The first OM attribute of the OM object, action-error[0], is created using the OM_OID_DESC macro. The OM object is terminated by the presence of the OM_NULL_DESCRIPTOR macro as the last element.

“Declaring OM Objects” on page 206 describes the remaining OM descriptors in an OM object, which correspond to the attributes defined in the OM class definitions in the *NetView for AIX Programmer's Reference*.

The Object-ID Convenience Routines

All attributes and objects have object identifiers (OIDs). In XMP, these OIDs are encoded as hexadecimal strings, which appear as OM attributes in several OM objects. You may need to manipulate an object-identifier string as you fill in or interpret an OM object.

There are six convenience routines provided to help you. These routines begin with the prefix at_ and are described in the following list:

at_array_to_oid()	Converts an array of integers corresponding to an object identifier to an object-identifier string.
at_free()	Releases dynamic memory previously allocated by other memory functions.
at_oid_to_array()	Converts an object-identifier string to an array of integers corresponding to the object identifier.
at_oid_match()	Determines whether two object-identifier strings match.
at_oid_to_str()	Converts an object identifier string to a NULL-terminated string in dot-notation form. This is the reverse operation of at_str_to_oid().
at_str_to_oid()	Converts a NULL-terminated string in dot-notation form to a BER-encoded object-identifier string.

Note: The at_free() routine should be called to release memory after using these object-identifier convenience routines.

Declaring OM Objects

This section describes how to derive a declaration from a class definition. There are two examples in this section.

Example Parameter for an XMP Function: Table 16 defines the OM class called CMIS-Get-Result. An object of this OM class is an input parameter to the mp_get_rsp() function. The mp_get_rsp() function is described later, but its basic purpose is to send the requested information about some managed object.

Table 16. OM Attributes of a CMIS-Get-Result OM Object

Value OM Attribute	Value Syntax	Value Length	Value Number	Initial Value
managed-Object-Class	Object(Object-Class)	-	0-1	-
managed-Object-Instance	Object(Object-Instance)	-	0-1	-
current-Time	String(Generalized-Time)	-	0-1	-
attribute-List	Object(Attribute)	-	0 or more	-

The first column gives the name of each OM attribute of the OM class; the second column gives the syntax of each OM attribute; the third column specifies the maximum length (if any) of a string-type OM attribute; the fourth column specifies the limits on how many instances of the OM attribute can be present, and the last column gives the default value of the OM attribute, if any.

The following code fragment uses the previous definitions to declare an OM object, called `_publicGetResultObj`, whose OM class is *MP_C_CMIS_GET_RESULT*. In a real application, the values of the attributes may be determined at run time.

```
OM_descriptor _publicGetResultObj[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_CMIS_GET_RESULT),
  {MP_MANAGED_OBJECT_CLASS, OM_S_OBJECT, {{0, _publicObjectClass}}},
  {MP_MANAGED_OBJECT_INSTANCE, OM_S_OBJECT, {{0, _publicObjectInstance}}},
  {MP_ATTRIBUTE_LIST, OM_S_OBJECT, {{0, _publicAttribute}}},
  OM_NULL_DESCRIPTOR
};
```

The OM attributes of `_publicGetResultObj` are constructed as follows:

MP_MANAGED_OBJECT_CLASS: The value of the attribute MP_MANAGED_OBJECT_CLASS must be set to an instance of the OM class OBJECT_CLASS that will contain the object identifier of the managed object class encoded in BER.

The following fragment of code shows a public object whose class is OBJECT_CLASS:

```
OM_descriptor _publicObjectClass[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_OBJECT_CLASS),
  {MP_GLOBAL_FORM, OM_S_OBJECT_IDENTIFIER_STRING,
  OM_STRING(OMP_O_DMI_O_EVENT_FORWARDING_DISCRIMINATOR)},
  OM_NULL_DESCRIPTOR
};
```

The value of the MP_GLOBAL_FORM attribute is the object identifier, in BER encoding, of the managed object class. This value can be obtained either through the convenience routines (at_str_to_oid() or at_array_to_oid()) or from the constants defined in the header file of the Management Contents Package.

The identifier OMP_O_DMI_O_EVENT_FORWARDING_DISCRIMINATOR is a macro which evaluates to a string corresponding to the encoded form of the object identifier 2.9.3.2.3.4. The macro OM_STRING is used to convert it to an OM_object_identifier.

MP_MANAGED_OBJECT_INSTANCE: The MP_MANAGED_OBJECT_INSTANCE OM object identifies (by Distinguishing-Name) the instance of the object on which the GET operation was performed. As described in the Chapter 11, “Understanding the NetView for AIX Management Environment” on page 167, a Distinguished-Name consists of a sequence of Relative-Distinguished-Names, each of which usually consists of one Attribute-Value-Assertion (AVA). Within the AVA, the MP_NAMING_ATTRIBUTE_ID is another OID in BER encoding format, which can be manipulated as described above for the managed object class.

```
/* Objects for building an Object Instance */

static OM_descriptor _publicDiscriminatorIdValue[] =
{
  OM_OID_DESC(OM_CLASS, C_DMI_SIMPLE_NAME_TYPE),
  {DMI_NUMBER, OM_S_INTEGER, {{1}}},
  OM_NULL_DESCRIPTOR
};

OM_descriptor _publicAvaObj[]=
{
  OM_OID_DESC(OM_CLASS, MP_C_AVA),
  {MP_NAMING_ATTRIBUTE_ID, OM_S_OBJECT_IDENTIFIER_STRING,
  OM_STRING(OMP_O_DMI_A_DISCRIMINATOR_ID)},
  {MP_NAMING_ATTRIBUTE_VALUE,OM_S_OBJECT, {{0,_publicDiscriminatorIdValue}}},
  OM_NULL_DESCRIPTOR
};

OM_descriptor _publicDsRdnObj[]=
{
  OM_OID_DESC(OM_CLASS, MP_C_DS_RDN),
  {MP_AVAS, OM_S_OBJECT, {{0, _publicAvaObj}}},
  OM_NULL_DESCRIPTOR
};

OM_descriptor _publicDsDnObj[]=
{
  OM_OID_DESC(OM_CLASS, MP_C_DS_DN),
  {MP_RDNS, OM_S_OBJECT, {{0, _publicDsRdnObj}}},
  OM_NULL_DESCRIPTOR
};

OM_descriptor _publicObjectInstance[]=
{
  OM_OID_DESC(OM_CLASS, MP_C_OBJECT_INSTANCE),
  {MP_DISTINGUISHED_NAME, OM_S_OBJECT, {{0, _publicDsDnObj}}},
  OM_NULL_DESCRIPTOR
};
```

Note that the syntax of MP_NAMING_ATTRIBUTE_VALUE OM attribute of the AVA is labelled *any*. When the OM class definition specifies a syntax of *any* for some OM attribute, follow these rules to fill-in the syntax and value fields:

1. If the attribute has a syntax that is defined in one of the packages (Common, CMIS, SNMP or Management Contents Packages), you can just use that syntax, and set the value accordingly.

2. Alternatively, if the value is an ASN.1 unconstructed type (as defined in the table following), you can use the appropriate OM syntax specification (prefixed with *OM_S_*, and set the value accordingly. The following OM syntaxes correspond to ASN.1 simple types:

   OM_S_BIT_STRING
   OM_S_BOOLEAN
   OM_S_GENERAL_STRING
   OM_S_GENERALIZED_TIME
   OM_S_GRAPHIC_STRING
   OM_S_IA5_STRING

OM_S_NUMERIC_STRING
OM_S_OBJECT_DESCRIPTOR_STRING
OM_S_OCTET_STRING
OM_S_INTEGER
OM_S_NULL
OM_S_OBJECT_IDENTIFIER_STRING
OM_S_PRINTABLE_STRING
OM_S_VIDEOTEX_STRING
OM_S_VISIBLE_STRING
OM_S_TELETEX_STRING
OM_S_UTC_TIME_STRING
OM_S_UNLIMITED_INTEGER
OM_S_REAL

3. If the attribute syntax is a constructed ASN.1 type, the syntax must be set to OM_S_OBJECT and the value must be an instance of the OM class that maps the ASN.1 constructed type. This is done with the attribute DMI_A_DISCRIMINATOR_ID whose value is an instance of class DMI_SIMPLE_NAME_TYPE.

4. Last, you have the option to set the syntax field to *OM_S_ENCODING_STRING*, and use your own functions to encode (or decode) the attribute value (using BER) to (or from) a string representation. This encoded string is used as the value of the attribute.

MP_ATTRIBUTE_LIST: The *OM_descriptor*s of type *MP_ATTRIBUTE_LIST* correspond to the attributes of the managed object. Each attribute of a managed object is represented by an OM object of OM class *MP_C_ATTRIBUTE*. In this example, the GET request has requested the value of one attribute of the managed object.

```
static OM_descriptor _publicAttributeId[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_ATTRIBUTE_ID),
  {MP_GLOBAL_FORM, OM_S_OBJECT_IDENTIFIER_STRING,
  OM_STRING(OMP_O_DMI_A_DISCRIMINATOR_ID)},
  OM_NULL_DESCRIPTOR
};

static OM_descriptor _publicDiscriminatorIdValue[] =
{
  OM_OID_DESC(OM_CLASS, C_OMI_SIMPLE_NAME_TYPE),
  {DMI_NUMBER, OM_S_INTEGER, {{1}},
};

static OM_descriptor _publicAttribute[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_ATTRIBUTE),
  {MP_ATTRIBUTE_ID, OM_S_OBJECT, {{0, _publicAttributeId}}},
  {MP_ATTRIBUTE_VALUE, OM_S_OBJECT, {{0,_publicDiscriminatorIdValue}}},
  OM_NULL_DESCRIPTOR
};
```

There is one point to note about an *MP_C_ATTRIBUTE* OM object:

- The *MP_ATTRIBUTE_VALUE* OM attribute is another point where the OM class definition specifies a syntax of *any*. See the discussion on the *MP_MANAGED_OBJECT_INSTANCE* in the previous section for details about what this means.

This example is a graphical summary of the relationships of various parts of one XMP *response*. Other *argument*s and *response*s are based on the same principles and have similar structure. Once you understand this example well, the other parameters are easier to follow.

About the Number of OM Descriptors: In general, each OM descriptor corresponds to one OM attribute from the formal definition of the OM class in the *NetView for AIX Programmer's Reference*. To be more precise, each OM descriptor corresponds to the *value* of one formal OM attribute from the OM class definition.

This clarification is most important for those OM class definitions with OM attributes that can have multiple values. In the class definition tables, such OM attributes have an *or more* statement in the value number column.

The MP_ATTRIBUTE_LIST OM attribute in the current example has multiple values. Thus, there may be multiple instances of this descriptor.

About the Order of OM Descriptors: To further clarify, an OM object consists, in general, of an *unordered* list of OM attribute values. That means that your declaration does not necessarily have to maintain the order of the OM class definition or template declaration file. The values for a multi-valued OM attribute are, however, ordered.

Note: The OM descriptors for a multi-valued OM attribute must be grouped together without intervening OM attributes. In fact, all OM descriptors for multi-valued OM attributes are always grouped together, regardless of their source.

Thus, when you receive a message containing one or more multi-valued attributes, you can be sure that all the values for each attribute are adjacent.

Your processing of an OM object must account for the fact that the list of OM descriptors in an OM object is unordered. You could use a loop that incorporates the following two steps:

- Determine the type of the next OM descriptor, probably in a switch block.
- If it is not OM_NULL_DESCRIPTOR, process it accordingly; otherwise, terminate the loop.

Alternatively, you could write an utility function that takes as input a pointer to an OM object and an OM attribute identifier, and returns an index (pointer) to the first corresponding OM descriptor in the OM object. This could be used to avoid the loop and switch.

Another Example for an XMP Parameter: Table 17 on page 211 illustrates a parameter to the mp_get_req() function, this time the SNMP version of a GET argument.

Table 17. OM Attributes of an SNMP-Get-Argument

Value OM Attribute	Value Syntax	Value Length	Value Number	Initial Value
responder-Ip-Address	Object(Network-Address)	-	0-1	-
var-Id-list	String(Object-Identifier)	-	1 or more	-
access-Control	Object(Access-Control)	-	0-1	-

The following code fragment shows the public objects that form an argument of the *mp_get_req()* function.

```
/*  Network Address */

OM_descriptor _publicNetworkObj[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_NETWORK_ADDRESS),
  {MP_IP_ADDRESS, OM_S_OCTET_STRING, {4, "\x9\x4F\x1\x6D"}},
  OM_NULL_DESCRIPTOR
};

/* Community name */
OM_descriptor _accessControl[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_COMMUNITY_NAME),
  {MP_COMMUNITY, OM_S_OCTET_STRING, OM_STRING("public")},
  OM_NULL_DESCRIPTOR
};

/* Declarations for snmp_get_argument */
OM_descriptor _publicGetArgumentObj[] =
{
  OM_OID_DESC(OM_CLASS, MP_C_SNMP_GET_ARGUMENT),
  {MP_RESPONDER_IP_ADDRESS, OM_S_OBJECT, {{0, _publicNetworkObj}}},
  {MP_VAR_ID_LIST, OM_S_OBJECT_IDENTIFIER_STRING, {{0,0}}},
  {MP_ACCESS_CONTROL, OM_S_OBJECT, {0, _accessControl}},
  OM_NULL_DESCRIPTOR
};
```

The XOM Functions

The following list summarizes each of the functions in the XOM library.

om_copy()	Creates an independent duplicate of a private OM object.
om_copy_value()	Copies a string from one private OM object to another.
om_create()	Creates a new private OM object of a particular class.
om_decode()	Creates a private OM object that represents an encoded private OM object.
om_delete()	Deletes a service-generated OM object.
om_encode()	Creates a new private OM object that encodes an existing private OM object, using the Basic Encoding Rules for ASN.1.
om_get()	Creates a public copy of all or part of a private OM object.
om_instance()	Tests an OM object for membership in a particular OM class.

om_put()	Puts attribute values of an OM object, public or private, into a private OM object.
om_read()	Reads a segment of a string from a private OM object.
om_remove()	Removes and discards an attribute value, or the entire attribute itself, from a private OM object.
om_write()	Writes a segment of a string into a private OM object.

Most of these functions have a reciprocal relationship with another function:

- om_delete() has the reverse effect of om_create().
- om_decode() has the reverse effect of om_encode().
- om_remove() has the reverse effect of om_put().
- om_read() has the reverse effect of om_write().

Refer to the *NetView for AIX Programmer's Reference* for a detailed description of each function.

The XOM functions are always used in concert with XMP functions. Examples of how the XOM functions are used appear together with examples of XMP functions in Chapter 13, “Using the XMP API” on page 213.

Chapter 13. Using the XMP API

XMP was designed to offer the services of both CMIS and SNMP. This section introduces XMP, its services, some common arguments and parameters, and other important aspects of XMP. Later sections provide more information on performing basic program tasks.

The Terminology of XMP

This chapter uses many of the terms that were defined in "Defining Network Management Systems" on page 167. Two new terms that are important in understanding XMP are defined here:

Responder	A responder is a reactive entity that responds to, or services, the (asynchronous) requests of one or more requesters. Most of the activities of an agent place it in the role of a responder; however, an agent takes the role of a requester when it issues a notification or event.
Requester	A requester is a proactive entity that requests data and services from one or more responders. Most of the activities of a manager place it in the role of a requester; however, a manager takes the role of a responder when it receives (asynchronously) a notification or event.

Both of these roles can be taken on by either a manager or an agent.

XMP C-language Naming Conventions

This section uses identifiers that are defined in the following header files:

- xom.h
- xmp.h
- xmp_cmis.h
- xmp_snmp.h

Table 18 shows the conventions used by XMP in assigning C-language identifiers.

Throughout this chapter, the full C-language identifier is used for the items in this table.

Table 18 (Page 1 of 2). XMP Interface Naming Conventions

Item	Convention	Example
Functions	The names of XMP functions are always prefixed with `mp_`, and are always in lower case letters. Also, many XMP functions have, as a suffix, either `_req()` or `_rsp()`. These indicate CMIS and SNMP requests and response functions, respectively. The empty parentheses also help distinguish function names in this document.	`mp_get_req()`

Table 18 (Page 2 of 2). XMP Interface Naming Conventions

Item	Convention	Example
Function parameters and results	The names of XMP function parameters and results are always in lower case, italicized, and have no special prefix. Since C-language functions have a single result, some XMP functions have parameters that point to additional return values. The names of such parameters have a suffix of _return. Thus, you can easily distinguish output parameters from input parameters that just happen to be pointers.	*argument* **result_return*
Data types	The names of data types are always prefixed with two capital letters, either OM_ or MP_, depending on whether they are defined by XOM or XMP, respectively. The rest of such names are always in lower case letters.	MP_status
Macros and defined constants	The names of XMP macros and defined constants, including XMP-defined OM attribute names, are always prefixed with MP_, and are always in upper case letters.	MP_IP_ADDRESS
OM class names	The names of XMP-defined OM class names are always prefixed with MP_C_, and are always in upper case letters.	MP_C_SNMP_GET_RESULT
Error-related constants	The names of the values for error-related enumerated types defined by XMP are always prefixed with MP_E_, and are always in upper case letters.	MP_E_INVALID_FILTER
Enumeration constants (tag values)	The names of the values for other enumerated types defined by XMP are always prefixed with MP_T_, and are always in upper case letters.	MP_T_COLD_START

The Protocol Services of XMP

XMP supports the seven CMIS services, through the CMIS OM package, and the four SNMP services, through the SNMP package. The services of the three shared services are mapped to shared function names, as shown in Table 19.

Table 19 (Page 1 of 2). XMP Functions to Support CMIS and SNMP Services

CMIS Service	SNMP Service	XMP Functions	Description (of request only)

Action	–	`mp_action_req()`
		`mp_action_rsp()`
			Requests that the responder perform one of the actions defined for an object.

Cancel get	–	`mp_cancel_get_req()`
		`mp_cancel_get_rsp()`
			Requests that the responder terminate servicing an earlier asynchronous get request that has not yet been completed.

Create	–	`mp_create_req()`
		`mp_create_rsp()`
			Requests that the responder create an instance (object) of the specified object class.

Table 19 (Page 2 of 2). XMP Functions to Support CMIS and SNMP Services

CMIS Service	SNMP Service	XMP Functions	Description (of request only)
Delete	–	`mp_delete_req()` `mp_delete_rsp()`	Requests that the responder destroy a particular instance (object) of an object class.
Get	Get	`mp_get_req()` `mp_get_rsp()`	Requests that the responder supply the values of one or more object attributes.
Set	Set	`mp_set_req()` `mp_set_rsp()`	Requests that the responder modify the values of one or more object attributes.
Notification	Trap	`mp_event_report_req()` `mp_event_report_rsp()`	Issues one of the notifications (events or traps) defined for an object.
–	Get next	`mp_get_next_req()` `mp_get_next_rsp()`	Requests that the responder supply the type (name) and value of the next SNMP variable in the object.

The requester functions have the `_req()` suffix. The responder functions have the `_rsp()` suffix. The responder functions are used only to respond to a request.

For the get, set and notification (trap) requests, the XMP functions accept protocol-specific parameters for both CMIS and SNMP.

The Supporting Functions of XMP

In addition to the protocol services, XMP defines several additional functions that are necessary to manage the environment and others that support asynchronous activity. These are described in Table 20 and Table 21 on page 216.

Table 20. XMP Functions to Manage the Environment

XMP Functions	Description
mp_bind()	Opens a session with the communications infrastructure. This function returns an MP_C_SESSION OM object, which is described in “Common Parameters and Results of XMP Functions” on page 216.
mp_error_message()	Maps an MP_status OM object into a null-terminated string that contains an error message describing the error.
mp_initialize()	Initializes the XOM workspace. This function returns a handle to the work-space.
mp_shutdown()	Discards the workspace. After this call, no XMP functions may be used until mp_initialize() initializes a new workspace.
mp_unbind()	Terminates a given session with the communications infrastructure. This function does not destroy the associated MP_C_SESSION OM object but invalidates it for further use.
mp_version()	Associates OM packages and features with the workspace that was initial-ized with mp_initialize().
mp_wait()	Waits for the availability of management messages from one or more bound sessions.

Table 21. XMP Functions to Support Asynchronous Activity

XMP Functions	**Description**
mp_abandon()	Abandons a pending asynchronous request. Any associated response is discarded without notice.
mp_receive()	Used to obtain the argument of an asynchronous message. Responders use it to receive requests and to receive responses to notifications. Requesters use it to receive notifications and to obtain the results from asynchronous requests.
mp_wait()	Waits for the availability of management messages from one or more bound sessions.

There are three convenience routines that you can use to register XMP-based applications to receive filtered events. These routines are described in “NetView for AIX Event Registration Routines” on page 312.

Common Parameters and Results of XMP Functions

This section describes the common parameters and results from a typical XMP function. The following example illustrates an XMP function call:

```
mp_get_req (session, context, argument, result_return, invoke_id_return)
```

OM objects of two important OM classes appear as input parameters to most XMP functions. They are:

- MP_C_SESSION
- MP_C_CONTEXT

The values of parameters of the OM class MP_C_SESSION are static over the life of the binding. The values of parameter of the OM class MP_C_CONTEXT are static over the life of one or more transactions (requests or notifications). They are collected in these OM objects to simplify the parameter list for XMP functions.

In addition, other data structures appear regularly as output or results of most of the XMP functions:

- *status* (one of several concrete subclasses of the abstract OM class Error)
- *invoke_id*

Names, Addresses, and Titles

XMP provides a variety of ways to identify the source or destination of a message. The terms used for this have very specific meanings in this context. Each form of identifier indicates an abstract OM class, whose subclasses are used as described in the following list:

name The subclasses of name are used to define managed object instances. For example, the subclass MP_C_DS_DN is used for ISO-naming of managed objects.

address The subclasses of address are used to define the specific location of a particular manager or agent. For example, the subclass MP_C_NETWORK_ADDRESS consists of an octet string containing an IP address that can be used to specify the location of an agent or manager.

title The concrete subclasses of title are used to define the specific system name or management process responsible for a managed-object instance. In particular, the string contained in the subclass MP_C_ENTITY_NAME corresponds to the name specified in the first line of the Local Registration File (LRF).

Address Specification

The way in which the source and destination of a message are specified depends on the protocol being used. In the case of SNMP, a responder address can be specified as a parameter in the argument for any particular request. In the case of CMIP, the communications infrastructure uses the ORS directory information to map the name of the managed object to the appropriate title or address.

Precedence in Access Control

Any access control specified in the session is overridden by access control specified in the context. Likewise, access control specified in the context is overridden by access control specified in the function argument.

The Session Parameter

Most XMP functions require a parameter called *session*, an OM object whose OM class is MP_C_SESSION. The mp_bind() function uses MP_C_SESSION OM objects as both input and output parameters. You can think of a session as a binding to the communications infrastructure. You can create multiple bindings to the communications infrastructure by making multiple calls to mp_bind(); each call returns a distinct *session*. This facilitates having sessions with differing characteristics. This technique might be used in a single process acting as multiple agents, or to tailor a binding for a particular destination.

Note: Once the mp_bind() function starts a session, you cannot modify any of the OM attributes of the session OM object. Doing so may cause unpredictable, potentially catastrophic errors.

You can use the default MP_C_SESSION OM object called MP_DEFAULT_SESSION as input to the mp_bind() function. Its characteristics are described in the Session class definition in the *NetView for AIX Programmer's Reference*.

Note: The default session is not an appropriate choice for a management program that has a responder role. The properties of the default session do not allow the session to receive indications, because it does not include the entity name. The NetView for AIX program requires the entity name to provide location transparency.

Function of the Session

The session serves several purposes:

Addressing: The *session* parameter can contain addressing information for both the requester and responder.

Establishing the Role of the Management Program: The *session* parameter includes an OM attribute, MP_ROLE, used to specify whether to enable manager or agent roles for the session.

Default Access Control: Most of the XMP routines have arguments which include an optional access-control attribute. When the underlying service provider uses CMIP, the program does not support access control for association establishment, so this attribute must not be present in session or context objects that will be used to access CMIP agents. When it is part of the request, such as the Community Name in an SNMP-Get-Argument or the External Access Control in a CMIS-Set-Argument, it is sent to the responder, and the responder is responsible for validating the Access Control information.

Process Handling: One OM attribute of the session, MP_FILE_DESCRIPTOR, provides a file descriptor that is used in the mp_wait() function. If the session is inactive, the value is MP_NO_VALID_FILE_DESCRIPTOR.

The Context Parameter

Most XMP functions require a parameter called *context*, which is an OM object whose OM class is MP_C_CONTEXT.

The *context* parameter defines several characteristics of a transaction that frequently are identical for a number of transactions. The *context* parameter collects these into a single OM object which can be created and reused as necessary.

The defined default context, MP_DEFAULT_CONTEXT, is always valid as input to XMP functions, though possibly insufficient for some management programs. Details about these default characteristics appear in the description of the context OM class in the *NetView for AIX Programmer's Reference*.

The OM class MP_C_CONTEXT contains two key groups of OM attributes:

- Service controls
- Local controls

Service Controls of the Context

The service control OM attributes are described in the following list:

MP_ACCESS_CONTROL
This optional OM attribute specifies access-control information that extends any access-control information specified in the *session* parameter.

MP_MODE
Requests either confirmed (MP_T_CONFIRMED) or non-confirmed (MP_T_NON_CONFIRMED) service. This OM attribute is meaningful only for calls to mp_set_req(), mp_event_report_req(), and mp_action_req().

MP_PRIORITY
This OM attribute is not used by the NetView for AIX program.

Local Controls of the Context

The local control OM attributes are described in the following list:

MP_ASYNCHRONOUS
: Indicates whether a function call should be made synchronously (OM_FALSE) or asynchronously (OM_TRUE) as described in "Synchronous and Asynchronous Calls in XMP" on page 221. Its value is meaningful only for those functions that can be called either way.

MP_SIZE_LIMIT
: Indicates the maximum number of linked responses allowed for any synchronous call to mp_get_req(), mp_set_req(), mp_action_req(), or mp_delete_req(). Details about the effect of exceeding this limit appear in the description of this OM class in the *NetView for AIX Programmer's Reference*.

MP_TIME_LIMIT
: Indicates the number of seconds to wait for a response to a synchronous function call. If this limit is exceeded, the request is terminated. Details about other effects appear in the description of this OM class in the *NetView for AIX Programmer's Reference*.

Other Input Parameters

Besides the *session* and *context* parameters, each protocol function in XMP has a function-specific input argument:

- For requester functions, the function-specific input argument is called *argument*.
- For responder functions, the function-specific input argument is called *response*.

Each of these is an OM object with OM attributes appropriate for the function. See the *NetView for AIX Programmer's Reference* for descriptions of the protocol functions of XMP and the appropriate OM class definitions.

Interpreting the Object Instance

When an agent receives a request, it must interpret the object instance and translate the object instance information into addressing information that will be used to access the object. For optimal performance, the agent should not have to translate the object instance information before using it as an addressing mechanism. To eliminate translation, use the object's distinguishing attribute to identify the object instance when you send requests from your management program. For example, a workstation agent can use an IP address for the distinguishing attribute. When this agent receives a request, it can access the specified object (workstation) by using the distinguishing attribute.

The distinguished name (DN) for a LANIC card on a workstation might be a concatenation of the network relative distinguished name (RDN), the LAN RDN, the workstation RDN, and then the LANIC RDN. In this case, the distinguishing attribute for the LANIC card would be the last RDN from the distinguished name.

However, in some cases the last distinguished name is not enough. For instance, imagine an agent that manages all of the LANIC cards on a LAN. To identify a specific object instance of a LANIC card, the agent needs the distinguished name of both the workstation and the LANIC card.

If an agent encounters an error interpreting the last part of the DN, it should verify the complete name. The assumptions made about the previous part of the Fully Distinguishing Name (FDN) may have been wrong.

If you cannot model a distinguishing attribute that can be used as the addressing mechanism for the object, the agent has to maintain an internal table that translates the object instance information into an address mechanism. The implementation of this is object-dependent.

Return Values for XMP Functions (Status)

The functional result of every XMP function is a private OM object, MP_status. This is one of the OM object constants (MP_SUCCESS, MP_INVALID_SESSION, MP_NO_WORKSPACE, MP_INSUFFICIENT_RESOURCES), or an instance of one of the five concrete subclasses of the abstract OM class Error. The details of these OM objects are described in “The OM Class Error” on page 228.

Invoke Identification Parameter

All the XMP protocol requester functions have an output parameter called *invoke_id_return*. This parameter is an integer that is different for every invocation of a requester function and uniquely identifies the particular call. It is provided solely to support asynchronous operation.

When XMP is used asynchronously, the result is obtained later in a call to mp_receive(). One of the output parameters of mp_receive() is the *invoke_id_return* parameter of the function call whose result is being returned.

You need to keep a record of *invoke_id_return* for every asynchronous call, so that you later can match the incoming result.

The Result Parameter

The XMP request functions, with the exception of mp_cancel_get_req(), include a parameter called *result_return*. This pointer points to a private OM object that contains the result of the request as sent by the responder. However, this is true only under the following conditions:

- The request function must be called in synchronous mode.
- The request function return value (MP_Status) must be MP_SUCCESS.

If the request is made asynchronously, encounters an error, or fails to initiate a transaction, the *result_return* parameter is not valid.

When the *result_return* parameter is valid, the OM object can be composed of either a single reply or multiple linked replies.

Linked Replies

Some requests can cause multiple linked replies. When a request causes multiple replies, each reply is called a partial result. The following CMIS requests cause multiple linked replies:

- mp_action_req()
- mp_delete_req()
- mp_get_req()
- mp_set_req()

For a synchronous call, the *result_return* parameter is a private OM object of OM class MP_C_MULTIPLE_REPLY; this is a list of OM objects of OM class MP_C_CMIS_LINKED_REPLY_ARGUMENT, each of which constitutes a partial result.

You can examine this *result_return* parameter by using XOM function calls, such as om_instance() and om_get().

For an asynchronous call, the *result_or_argument_return* value from a call to mp_receive() is an OM object of OM class MP_C_CMIS_LINKED_REPLY_ARGUMENT.

Notes:

1. When an agent receives a request for a managed object, it must respond. If it does not recognize the specified object, it must return the constant MP_ABSENT_OBJECT. For a scoped request, the agent must signal the end of replies by returning MP_ABSENT_OBJECT in the final message.
2. Be sure to identify the type of the attribute present in the linked reply argument, because many attributes that contain different OM classes are permitted in this object.

Synchronous and Asynchronous Calls in XMP

As a programmer, you are familiar with synchronous function calls. Code that follows a synchronous function call can assume that the effect of the call is complete. For example, when the malloc() function returns, memory has (or, for errors, has not) been allocated. This call to malloc() will not result in memory allocation at some indeterminate time in the future. The call and its effect are synchronous.

XMP enables you to make any call, except mp_cancel_get_req(), synchronously, and to use requester functions asynchronously. When you make an asynchronous function call, the XMP interface first determines if the call is valid. If so, the transaction with the responder is initiated, and your management program is notified, through MP_Status, that the call succeeded. Success in this case applies to the successful initiation of the request; the request itself has not yet been processed.

After your management program is notified of the successful initiation of the request, it can continue processing while the request is being serviced. If a response is expected, you must later call mp_receive() to determine the outcome of the request. If the responder was unable to process the request successfully, that fact becomes known when the response is processed.

Requester function calls are made synchronously or asynchronously based on the value of the Boolean MP_ASYNCHRONOUS OM attribute of the *context* parameter. If this attribute is set to OM_TRUE, requester function calls are made asynchronously; otherwise, they are made synchronously.

All XMP responder functions and all XMP supporting functions are always synchronous; there is no asynchronous option for them.

Synchronous Requester Operation

When you make synchronous requester calls, use the following rules to interpret the effect of each call:

- If the return value of the function is MP_SUCCESS, the function call successfully initiated the request. If a reply was expected, the following points are also true:
  - The request was successfully processed.
  - The *result_return* parameter contains one of the private OM objects specified for the operation. For instance, in the case of a synchronous call to mp_get_req(), the result_return for a CMIS get operation contains an OM object whose OM class is MP_C_CMIS_GET_RESULT, MP_C_SNMP_GET_Result, MP_C_CMIS_MULTIPLE_REPLY, or MP_ABSENT_OBJECT (if no managed objects were selected for the request).

    In the case of a MP_C_CMIS_MULTIPLE_REPLY, each associated MP_C_CMIS_LINKED_REPLY_ARGUMENT OM object contains one reply. This reply may indicate an error, so you need to check all the individual replies to be sure that the operation was completely successful. The MP_C_CMIS_MULTIPLE_REPLY OM object terminates with an OM_NULL_DESCRIPTOR.
- If the return value of the function (its MP_status OM object) is not MP_SUCCESS, the request has failed, at least partially. The return value OM object is one of the concrete subclasses of the abstract OM class Error. This OM object indicates the nature of the error.

Asynchronous Requester Calls

The constant MP_MAX_OUTSTANDING_OPERATIONS specifies the maximum number of asynchronous transactions that can be outstanding at any time. An asynchronous transaction is outstanding until one of the following occurs:

- The last reply is received through mp_receive().
- The transaction is abandoned through mp_abandon().
- The session is closed through mp_unbind().
- The workspace is destroyed through mp_shutdown().

Because an asynchronous function call returns before the request has been serviced, you need a mechanism to obtain the responses to that request. In XMP, this mechanism is the mp_receive() function.

When you use asynchronous requester calls, use these rules to interpret the effect of each call:

- If the return value of the function, which is an MP_status OM object, is MP_SUCCESS, the function call was valid and successfully initiated the request. There is no information at this point about the success or failure of that request, only that it has been successfully initiated.

  The *invoke_id_return* parameter is a unique number that identifies this particular call. Use this identifier to match an incoming reply, from mp_receive(), with the call to which it pertains.

  On an asynchronous request, the *result_return* parameter is not valid. The result is obtained later through mp_receive().

- If the return value of the function is not MP_SUCCESS, the function call did not initiate a transaction; no reply is sent, even for a confirmed request.

  Note: If you have outstanding asynchronous calls on a session, wait for them to terminate and obtain their results before you issue a synchronous call on the session. You cannot mix synchronous and asynchronous calls on a session.

Handling Errors Reported by mp_receive()

If a responder encounters an error in servicing a request, it builds an appropriate *response* input parameter to the response function.

From the requester's point of view, certain output parameters from mp_receive() communicate the same information as the *MP_status* and the *result_return* parameter of a synchronous call. This correspondence is shown in Table 22.

Table 22. Comparing Parameters from Synchronous and Asynchronous Calls

Asynchronous parameters from mp_receive()	**Synchronous parameters**
operation_notification_status_return	MP_status (the function return value)
result_or_argument_return	result_return

The following rules apply to these mp_receive() output parameters:

- The *result_or_argument_return* parameter contains one of the private OM objects specified for the operation, if the operation_notification_status_return value is MP_SUCCESS. For instance, in the case of an asynchronous call to mp_get_req(), the *result_or_argument_return* parameter contains an OM object whose OM class is either MP_C_CMIS_GET_RESULT, MP_C_CMIS_MULTIPLE_REPLY, or MP_ABSENT_OBJECT (if no managed objects were selected for the request).

  If a CMIS request was scoped so that multiple objects were selected, several calls to mp_receive() report a MP_C_LINKED_REPLY_ARGUMENT response, each containing one reply from an agent. This reply may indicate an error so you need to check all the individual replies to be sure that the operation was completely successful. The agent signals the end of replies by sending MP_ABSENT_OBJECT as the last reply.

- If the *operation_notification_status_return* OM object is not MP_SUCCESS, the request failed, at least partially. The OM object is one of the concrete subclasses of the abstract OM class Error; which one determines the nature of the error. In this case, the result_or_argument_return is not valid.

Matching Responses to Outstanding Asynchronous Requests

An *invoke_id_return* value is returned by an asynchronous request. Later, when the response for that request is received, the mp_receive() function returns the same value in its own *invoke_id_return* parameter.

This gives you a mechanism to match the call to its reply. You, of course, must implement a scheme for recording invoke identification values and for matching them to your outstanding asynchronous requests.

Asynchronous Responder Operation

The role of a responder has no asynchronous characteristics. All responder activity is carried out synchronously. However, a responder must behave correctly to support asynchronous communications.

Receiving Requests and Notifications

Responder management programs receive requests, which may be request functions or notifications, through mp_receive(), but the role is reversed compared to that of a requester. If the request, such as mp_set_req(), was made in confirmed mode, you need to respond by calling the corresponding response function, such as mp_set_rsp().

When you receive a request or notification, such as a get request or event notification, one of the output parameters of the mp_receive() function is *invoke_id_return*. This value must be used as one of the input parameters to the response function, *invoke_id*. Do not modify the *invoke_id_return* value that you receive from mp_receive(), and ensure that the value you use in a response function matches the value from the corresponding call to mp_receive(). The invoke identification is the only link a requester has between its request and your response. If this link is lost, the response is discarded and the corresponding request eventually times out.

Abandoning Asynchronous Operations

XMP provides two functions that enable you to terminate an outstanding asynchronous request:

- The mp_abandon() function requests that XMP discard any further responses to the request. This has no effect on the activity of the responder.
- The mp_cancel_get_req() function is a confirmed service that sends a message to the responder, informing it that further processing of the get request is unnecessary. The responder is then free to terminate such processing.

  Using this function in a management program is beneficial, because it can reduce management traffic on the network. Locally, the effect is identical to that of mp_abandon().

Program Sequencing

This section provides information about the typical sequences of XMP function calls.

XMP Workspaces

The first step to take in creating a program that uses the XMP and XOM APIs is to create an *XMP workspace*. The XMP workspace is used by XOM to store data structures that it manipulates to support XMP functions. To create an XMP workspace, use the following statement as one of the first executable statements at the beginning of your management program:

```
workspace = mp_initialize();
```

The workspace should be discarded at the end of the management program, using the following statement:

```
mp_shutdown(workspace);
```

Program Initialization and Shutdown Sequences

Management programs that use XMP have characteristic initialization and shutdown sequences that involve five XMP functions:

- mp_initialize()
- mp_version()
- mp_bind()
- mp_unbind()
- mp_shutdown()

Initialization Sequence

The following steps apply to the initial phases of every management program operation:

Step 1. Create a workspace with mp_initialize(). This initial workspace supports only the Common and OM packages.

Step 2. Request that additional features be added to the workspace, using mp_version(). The *feature_list* parameter of this call normally contains either MP_CMIS_PACKAGE or MP_SNMP_PACKAGE and any management contents packages, such as DMI. This is dependent on your management program requirements and design. Use OM_EXPORT to export these identifiers in the global declarations. The OM_EXPORT XOM macro is described in "Using the OM_EXPORT and OM_IMPORT Macros" on page 204. At this point, you are able to create and manipulate OM objects using XOM functions and the class definitions in the packages you added.

Step 3. Open one or more sessions with the communications infrastructure using mp_bind(). The *session* OM object you get back from each call to mp_bind() is an input parameter to many XMP functions and contains common values for transactions. The session OM object is described in "The Session Parameter" on page 217.

At this point, your management program is in operational status and can perform any required activities in its role as an agent or manager.

Shutdown Sequence

The following steps apply to the closing phases of operation:

Step 1. Close all open sessions, using mp_unbind() on each one. This step abandons all outstanding asynchronous operations associated with each session. If you want to resolve outstanding requests, your management program can call mp_receive() repeatedly until the value of the *completion_flag_return* parameter is MP_T_NOTHING before issuing the mp_unbind() function call.

Step 2. Discard the workspace using mp_shutdown(). Note that a shutdown alone does not recover the resources associated with service-generated public OM objects created during the life of the workspace. Therefore, use om_delete() against each such OM object when you are finished with it.

Note: The function call mp_shutdown() automatically unbinds all sessions.

Operating Sequences

In general, every transaction (request or notification) between a manager and an agent begins in the idle state. The state of the transaction changes when an activity occurs, such as issuing or receiving a request. Later activity that resolves the current state causes the state to return to idle.

Note that these diagrams show the states of a particular transaction, not the global state of the process. Therefore, the idle state might well be renamed non-existent: transitions out of this state imply the initiation of a transaction; transitions into it imply the completion. Table 23 shows the typical activities of a management program in the role of a manager. Note that each line in the second column of the table is a separate trigger for the transition noted in the first column. Also, the number in parentheses refers to a different but related transition.

Table 23. State Transitions for a Manager

Transition Number	XMP Function

1	mp_action_req()
	mp_create_req()
	mp_delete_req()
	mp_get_req()
	mp_get_next_req()
	mp_set_req()

2	mp_receive() *(confirmation for (1))*
	Request (1) was synchronous and unconfirmed
	mp_abandon() *(request (1))*

3	*Receive partial (linked) reply*
4	mp_cancel_get_req() *(request (1))*
5	mp_receive() *(a notification)*
6	mp_event_report_rsp() *Notification (5) was unconfirmed*

Table 24 illustrates typical activities of a management program in the role of an agent:

Table 24. State Transitions for an Agent

Transition Number	XMP Function
1	mp_receive() *(an indication)*

2	mp_action_rsp()
	mp_cancel_get_rsp()
	mp_create_rsp()
	mp_delete_rsp()
	mp_get_rsp()
	mp_set_rsp()
	Indication (1) was unconfirmed

3	*Send partial (linked) reply*
4	mp_event_report_req()

5	mp_receive() *(confirmation for (4))*
	Request (4) was synchronous
	Request (4) was unconfirmed
	mp_abandon() *(request (4))*

You must ensure that every activity with which your management program is involved, as either requester or responder, returns to the idle state. To do otherwise represents incomplete handling of process interactions.

Important Note: Calls to the XMP/XOM library do not protect themselves from interrupt by signals. If your application uses signals, you *must* block signal delivery around all XMP/XOM function calls.

Handling Errors

This section summarizes information about managing errors that can arise in the XMP and XOM interfaces.

Errors are reported to your management program through the function return value. The XMP functions return, as their functional result, a private OM object, a member of the abstract OM class Error. More specifically, any particular function return value is a private OM object whose OM class is one of the concrete subclasses of the abstract OM class Error. Most commonly, XMP functions return the constant OM object MP_SUCCESS, which indicates a successful completion.

General Error Constants

XMP functions usually return the constant OM object MP_SUCCESS, which is self-explanatory.

The following three error constants (OM objects) can be returned by all XMP functions except mp_error_message() and mp_initialize():

MP_NO_WORKSPACE
: The workspace parameter is invalid.

MP_INVALID_SESSION
: The *session* parameter is invalid.

MP_INSUFFICIENT_RESOURCES
: There are inadequate local resources, such as insufficient memory, to execute the function.

The OM Class Error

Figure 16 illustrates the OM class Error and its subclasses. Abstract OM class names contain lower-case letters; concrete OM class names do not.

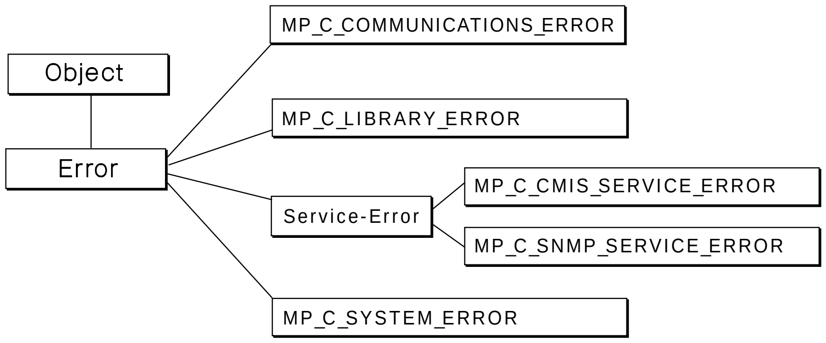

Figure 16. The Error OM Class

There are five concrete subclasses of the abstract OM class Error:

MP_C_COMMUNICATIONS_ERROR
: OM objects of this OM class report failures detected by the XMP library.

MP_C_LIBRARY_ERROR
: OM objects of this OM class report failures detected by the XMP library.

MP_C_CMIS_SERVICE_ERROR
: OM objects of this OM class report failures detected by the CMIS protocol.

MP_C_SNMP_SERVICE_ERROR
: OM objects of this OM class report failures detected by the SNMP protocol.

MP_C_SYSTEM_ERROR
: OM objects of this OM class report failures detected by the operating system.

The OM class Error includes two OM attributes inherited by all its subclasses:

MP_PROBLEM
: The syntax of this OM attribute is OM_S_ENUMERATED. Its value, prefixed by MP_E_, is a description of the difficulty, for example, MP_E_SIZE_LIMIT_EXCEEDED.

MP_PARAMETER
: The syntax of this OM attribute varies according to the MP_PROBLEM of the error. The OM class descriptions in the *NetView for AIX Programmer's Reference* explain, for each value of MP_PROBLEM, what the OM syntax of the associated MP_PARAMETER is.

  For some values of MP_PROBLEM, there is no associated MP_PARAMETER; in this case, the OM attribute is missing.

None of the subclasses of Error specify any additional OM attributes; however, they do specify particular values for these OM attributes.

For example, the MP_C_LIBRARY_ERROR OM class specifies 20 different values for MP_PROBLEM; an OM object of this OM class must have one of these values, some of which further specify the OM syntax of an associated MP_PARAMETER.

On the other hand, the MP_C_COMMUNICATIONS_ERROR OM class specifies different values for MP_PROBLEM and specifies an MP_PARAMETER with a syntax of OM_S_INTEGER.

A communications error occurs only when the communications infrastructure cannot send the message to the proper recipient. A communications error usually means either that the recipient (agent/manager) is not running, or that the agent has not been registered in the ORS.

The man page for each XMP function defines the errors it can return.

Note: Your management program never creates an OM object of any subclass of Error except the ones derived from Service-Error. OM objects of these OM classes are created exclusively by the XMP implementation as function return values, and, in the case of mp_receive(), as an output parameter.

Using XMP and XOM

As you begin coding management programs with these APIs, follow these steps:

Step 1. Determine the values that you want the *session* and *context* OM objects to have. These OM objects are described in “The Session Parameter” on page 217 and “The Context Parameter” on page 218. Use XOM functions, or suitable constants, to set these parameters.

Step 2. Initiate your workspace and session using mp_initialize(), mp_bind(), and mp_version() as described in “Initialization Sequence” on page 225. Perform any necessary local initialization.

At this point, a management program performing an agent role typically drops into a loop. The agent stays in the loop while waiting for an incoming request by using mp_wait(). When an incoming request arrives, the management program uses mp_receive() to get the request, services it, and returns to mp_wait(). The agent role requires that the management program be able to detect the conditions that require the issuance of an SNMP trap or CMIS notification, through mp_event_report_req(). How the management program performs the agent role is dependent on the management program design.

A management program that performs the manager role typically begins interacting with a user. Based on user input or other criteria, the management program issues requests to appropriate managed objects. The manager role can require that the management program receive and process SNMP traps and CMIS notifications, using mp_receive().

Note: A manager must receive incoming data as its top priority. An agent may issue many linked replies to a get request. If the manager responds slowly, or not at all, its interface with the postmaster may overflow, causing lost data.

Step 3. At each step, your design dictates when your management program needs to issue protocol requests and responses, such as mp_get_req(), mp_get_next_rsp(), or mp_event_report_req(). Refer to the *NetView for AIX Programmer's Reference* for the syntax and parameters of each XMP function you want to use.

Step 4. Use the XOM calls to build an appropriate argument OM object. This is highly dependent on the operation and managed objects involved. Use the object definition to find the registration identifications of the attributes in which you are interested and the OM class definition of the argument.

Step 5. Make the XMP call. Many requests can be made either synchronously or asynchronously. These modes are described in “Synchronous and Asynchronous Calls in XMP” on page 221.

The binding to the C language is explained in Chapter 14, “C-Language Binding for the XOM and XMP APIs” on page 231.

Example Programs

Example programs provide detailed illustrations of XMP calls with both protocols, CMIP and SNMP. The source files of the examples are located in the directory /usr/OV/prg_samples/xmp. To run these examples, you must log in as the root user.

To run the CMIP example, change directory to /usr/OV/prg_samples/xmp/cmip. Read the /usr/OV/prg_samples/xmp/cmip/README file, which explains how to run the example and what results to expect.

To run the SNMP example, change directory to /usr/OV/prg_samples/xmp/snmp. Read the /usr/OV/prg_samples/xmp/snmp/README file, which explains how to run the example and what results to expect.

A third example illustrates the use of the ovesmd daemon with the XMP API. To run this example, change directory to /usr/OV/prg_samples/xmp/ems. Read the /usr/OV/prg_samples/xmp/ems/README file, which explains how to run the example and what results to expect.

Chapter 14. C-Language Binding for the XOM and XMP APIs

This chapter explains certain characteristics of the C-language binding to the XMP API. The binding specifies C-language identifiers for all the elements of the XMP API so that management programs written in the C language can access the management information services. These XMP API elements include function names, typedef names, and constants.

There is a complete list of all the identifiers in the following header files:

- The xom.h file contains definitions for the X/Open OSI-Abstract-Data Manipulation (XOM) API.
- The xmp.h file contains common definitions for the XMP API.
- The xmp_cmis.h file contains specific definitions for the abstract services of the Common Management Information Service (CMIS) and the ASN.1 productions of the related protocol (CMIP).
- The xmp_snmp.h file contains specific definitions for the abstract services of the Simple Network Management Protocol (SNMP) and the associated ASN.1 productions.
- The header files for the Management Contents packages.

C-Language Naming Conventions

The XMP API uses part of the C-language public namespace for its facilities. All identifiers start with one of the following prefixes:

- mp
- MP
- OMP

The following prefixes are reserved and cannot be used by developers of management programs:

- mpP
- mpX
- MPX
- OMP
- _mp

The XOM API uses naming conventions that are similar, although not identical, to the XMP API naming conventions. The XOM API naming conventions are described in the *NetView for AIX Programmer's Reference*. All the XOM identifiers start with the prefix OM or om.

To improve readability, this book uses language-independent names for most of the API elements; however, the C-language name is given for the elements listed in Table 25:

Table 25. Elements for which C Language Name Is Provided

Element	**Description**	**Example C-language Name**
Function calls	The names of XMP functions have the prefix mp_ and are in lower-case letters. Also, many XMP functions have the suffix _req() or _rsp() to indicate CMIS request and response functions. Function calls end with parentheses.	mp_get_req()
Convenience rou-tines	The names of convenience routines have the prefix at_ and are in lower-case letters.	at_oid_to_str()
Function parameters	The names of XMP function parameters are in lower-case letters, italicized, and have no special prefix.	*name*
Data types	The names of data types have the prefix OM_ or MP_, depending on whether they are defined by XOM or XMP. The rest of the data type name is in lower-case letters.	MP_status

For all other elements, this book uses language-independent names.

Deriving C-Language Names from Language-Independent Names

The C-language names can be derived from the language-independent names by a mechanical process that depends on the kind of name. The following sections explain how to derive the C-language name from the language-independent name.

OM Class Names

To derive the C-language name for an XMP-defined OM class name, follow these steps:

Step 1. Change all letters to upper case
Step 2. Add the prefix MP_C_
Step 3. Change hyphens (-) to underscores (_).

Note: The language-independent OM class names are those used in ASN.1 with the exception that names containing multiple words are separated with hyphens.

Example: Get-Result becomes MP_C_GET_RESULT.

Defined Constants

To derive the C-language name for a macro or defined constant, including XMP-defined OM attributes but excluding error-related constants and enumeration constants, follow these steps:

Step 1. Change all letters to upper case
Step 2. Add the prefix MP_
Step 3. Change hyphens (-) to underscores (_)

Notes:

1. The name of an OM attribute is local to its OM class, which means that the same name may appear in different OM classes. For example, the OM attribute *filter* is defined in both the Get-Argument OM class and the Set-Argument OM class.

2. The language-independent OM attribute names are those used in ASN.1 with the exception that names containing multiple words are separated with hyphens.

Example: *scope* becomes MP_SCOPE.

Error Constants

To derive the C-language name for an error constant (the name of a value for an error-related enumerated type defined by XMP), follow these steps::

Step 1. Change all letters to upper case
Step 2. Add the prefix MP_E_
Step 3. Change hyphens (-) to underscores (_)

Example: *access-denied* becomes MP_E_ACCESS_DENIED.

Enumeration Constants

To derive the C-language name for an enumeration constant (tag value), follow these steps:

Step 1. Determine the value
Step 2. Change all letters to upper case
Step 3. Add the prefix MP_T_
Step 4. Change hyphens (-) to underscores (_)

Example: Enum(CMIS-Sync) becomes MP_T_BEST_EFFORT.

OM Attribute Limits

In the OM attribute tables in "Declaring OM Objects" on page 206, there are two columns labeled `Value Length` and `Value Number`. If the upper limit in either of these columns is not 1, it is given an identifier. To derive the C-language name for the upper limit for the value length or value number of an OM attribute, follow these steps:

Step 1. To the OM attribute name, add the prefix MP_VL_ for value length or MP_VN_ for value number.

Step 2. Change hyphens (-) to underscores (_).

Examples: the C-language name for the upper limit for the length of the value of the OM attribute IP-ADDRESS is MP_VL_IP_ADDRESS. The C-language name for the upper limit for the number of values of the OM attribute SUBSTRINGS is MP_VN_SUBSTRINGS.

Object Identifiers

To derive the C-language name for an object identifier, follow these steps:

Step 1. Change all letters to upper case

Step 2. Add the appropriate package identification and C prefix for the item, for example, MP_C_ for an OM class. The C prefixes are:

- C_ - for OM classes
- O_ - for Managed Objects
- A_ - for Attributes and Attribute Groups
- N_ - for Notifications
- S_ - for Parameters
- B_ - for Name Bindings
- P_ - for Packages

Step 3. Add the prefix OMP_O_

Step 4. Change hyphens (-) to underscores (_)

Table 26 gives some examples:

Table 26. Constructing C-language Names for Object Identifiers

Independent Name	C-language Name
top	OMP_O_DMI_O_TOP
activeDestination	OMP_O_DMI_A_ACTIVE_DESTINATION
communicationsAlarm	OMP_O_DMI_N_COMMUNICATIONS_ALARM
miscellaneousError	OMP_O_DMI_S_MISCELLANEOUS_ERROR
additionalInformation	OMP_O_DMI_P_ADDITIONAL_INFORMATION
discriminator-system	OMP_O_DMI_B_DISCRIMINATOR__SYSTEM

Function Return Value and Returned Parameters

If a function succeeds, the function return value is expressed in the C language by the constant MP_SUCCESS. If the function is not successful, an error is returned.

Because the C language does not provide multiple return values, functions must return all other results by writing into storage passed by the management program. Any argument that is a pointer to such storage has a name ending with *_return*. For example, the C-language parameter declaration *Uint * completion_flag_return* in the mp_receive() function indicates that the function will return an unsigned integer as a result; the actual argument to the function must be the address of a suitable variable. This notation enables the user to distinguish between an input parameter that happens to be a pointer and an output parameter where the asterisk is used to simulate the semantics of passing by reference.

Errors

If a function is not successful, it returns a value other than MP_SUCCESS and does not update the return parameters. The function return value in this case is one of the following:

- The constant MP_NO_WORKSPACE
- The constant MP_INVALID_SESSION
- The constant MP_INSUFFICIENT_RESOURCES
- A private object of one of the subclasses of the OM class Error

In order to allow automatic connection management (connection establishment and release that are not apparent), the XMP API might not communicate with a management program when mp_bind() is called. Instead, the XMP API might wait to establish the connection until a management operation or management notification is requested. Because of this flexibility, all functions can return the same errors as mp_bind(). For example, a get operation can return an authentication error because the connection was deferred until access actually was needed.

Supplied Parameters

If the value of a parameter to a function is not valid, such as a value outside the domain of the function or a pointer outside the address space of the program (or a null pointer), the function's behavior is undefined, unless otherwise stated in the function description.

Asynchronous Operations

Error reporting is more complicated for asynchronous operations, because these operations can fail either before the remote operation is started or during the remote operation. An error in the first stage is reported immediately in the return value of the invoking function. An error in the second stage is returned as the *operation_notification_status_return* result of a later call to mp_receive().

Compiling and Linking

All management programs that use the XMP API must include the xom.h and xmp.h headers, in that order, and at least one of the xmp_cmis.h and xmp_snmp.h headers.

When compiling management programs, no special switches are required. The following is an example of a program compile statement:

```
cc -c agent.c
```

A management program must be linked with the libxmp.a library. This library includes the following management services and packages:

- XOM API
- XMP API
- Common
- XOM
- LNV

A management program can be dynamically linked to the CMIS and SNMP management service packages by using the mp_version() function. No special link statement is required.

The following is an example of a program link statement:

```
cc agent.o -l xmp -o agent
```

Chapter 15. Introduction to the NetView for AIX WinSNMP API

This chapter describes the NetView for AIX implementation of the *Windows SNMP Manager API Specification Version 1.1a*. The specification represents the successful collaboration of many companies within the computer industry. Although originally drafted for the Microsoft Windows** environment, the design of the API is platform independent. IBM chose to implement NetView for AIX Windows SNMP (from here on referred to as WinSNMP) in its continued support for open technology. Parts of this chapter contain text from the specification. If you would like the complete specification document, see “SNMP Information” on page 383.

The original Internet-standard network management framework, as described in RFCs 1155, 1157, and 1212, is termed the SNMP version 1 framework (SNMPv1). In addition, there are three proposed Internet standards, as described in RFCs 1418, 1419, and 1420, that address the use of transports other than User Datagram Protocol (UDP) over Internet Protocol (IP) for SNMPv1. These RFCs describe SNMPv1 over Open Systems Interconnection (OSI), AppleTalk, and Internetwork Packet Exchanger (IPX).

The latest Draft-standard network management framework, as described in RFC 1901 through RFC 1908, is termed the Community-based SNMPv2 framework (SNMPv2C). (See “SNMP Information” on page 383 for a listing of RFCs.)

The *Windows SNMP Manager API Specification* introduces no constraints on the use of SNMPv1 or SNMPv2C, nor on the functionality supported by those protocols as prescribed in the relevant Internet RFCs. For the purposes of this document, SNMPv1 is seen as a subset of SNMPv2C.

What Is WinSNMP?

WinSNMP provides a single interface for application programmers and software vendors. This interface defines the procedure calls, data types, data structures, and associated semantics to which an application developer can program and which an SNMP software vendor can implement.

Figure 17 on page 238 shows IBM's WinSNMP end-to-end SNMP connectivity from an entity acting in a manager role to an entity acting in an agent role. This diagram is a high-level rendition of the model embodied in the current version of WinSNMP.

WinSNMP Architecture

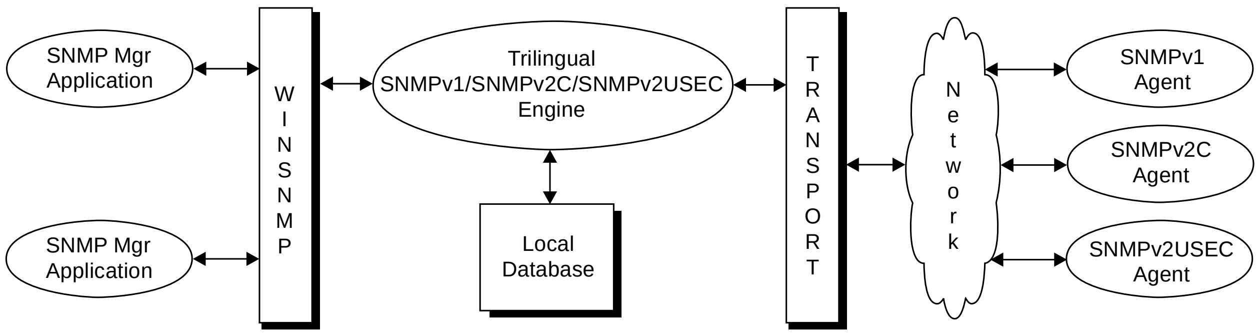

Figure 17. NetView for AIX WinSNMP Architecture

Benefits Provided by WinSNMP

WinSNMP offers these major benefits—all intended to accelerate the development, dissemination, and use of SNMP network management applications:

- SNMP enabling technology for functional network management applications (for example, SNMP makes ASN.1, BER, and SNMP protocol details transparent).
- SNMP service provider independence. A WinSNMP application runs with any compliant WinSNMP implementation.
- Uniform SNMPv1 and SNMPv2 support. A WinSNMP application does not have to know the SNMP version level of the target SNMP entities acting in an agent role. The WinSNMP implementation performs any and all necessary mappings between SNMPv1 and SNMPv2 in accordance with the appropriate RFCs.
- Transparent support for the secure SNMPv2 “User Based Security” model (SNMPv2USEC).

Compliance

Software which conforms to the *Windows SNMP Manager API Specification* is considered to be WinSNMP compliant.

Suppliers of implementations which are WinSNMP compliant are referred to as WinSNMP suppliers. Nothing in the *Windows SNMP Manager API Specification* is meant to dictate or preclude particular implementation strategies. The specification allows for various overlapping levels of SNMP support on the part of an implementation:

- Level 0 = Message encoding/decoding only
- Level 1 = Level 0 + interaction with SNMPv1 agents
- Level 2 = Level 1 + interaction with SNMPv2 agents
- Level 3 = Level 2 + interaction with other SNMPv2 managers

To be WinSNMP compliant, a vendor must implement 100% of the *Windows SNMP Manager API Specification*, as appropriate to the level of SNMP interaction the given implementation supports. WinSNMP vendors are encouraged to state clearly the level of SNMP interaction they support in all of their marketing and technical literature.

Note: The NetView for AIX WinSNMP API provides Level 3 implementation.

Applications that are capable of operating with any WinSNMP compliant implementation, which supports at least the level of SNMP interaction required by the application, are considered to have a WinSNMP interface and are referred to as WinSNMP applications.

The current version of the *Windows SNMP Manager API Specification* defines the use of the API by management applications. A future revision or separate extension may include features for use by SNMP agents. A companion document, the *WinSNMP/MIB API Specification*, provides definitions of elements used as operands to SNMP operations. If you would like the complete specification document, see "SNMP Information" on page 383.

SNMP

SNMP is a request-response protocol used to transfer management information between entities acting in a manager role and entities acting in an agent role. Managers are often configured as management stations and agents are often configured as managed nodes. A manager can also act as an agent to another manager in both vertical (hierarchical) and horizontal (distributed) relationships. Likewise, a physical node might be managed by multiple agents, and an agent might manage multiple physical nodes. When the prototypical management station/managed node perspective is used for the sake of simplicity and clarity of presentation, that practice is not meant to preclude other forms of SNMP interactions.

Each managed device or application contains monitoring and (possibly) control instrumentation. This instrumentation is accessed by the agent. The agent represents its access to this instrumentation to the manager through a Management Information Base (MIB), filtered by the SNMP security mechanisms. Management applications communicate with agents through SNMP to monitor and (possibly) control managed devices or applications.

A management application may issue several requests to an agent, without waiting for a response. Alternatively, it may issue a request and wait for a response, operating in a lock-step fashion with the agent. Furthermore, SNMP may be implemented on a wide range of transport protocols, each with varying delivery mechanisms and reliability characteristics. The normal transmission mechanism (UDP) is through non-guaranteed messages which may be dropped, duplicated, or re-ordered. Thus, with SNMP, it is the responsibility of each management application to determine and implement the desired level of reliability for its communications. This means that the management application decides on its own retransmission and timeout strategy.

An agent may send asynchronous messages, called traps in SNMPv1 or notifications in SNMPv2, to a management application. This important feature of SNMP is also fully supported by WinSNMP.

Note: In this document, the term traps is used to refer both to traps and notifications, unless specifically qualified in a given instance.

SNMP can be broken down into these distinct parts:

- Entities

Network nodes with the capability of transferring network management information using SNMP.

- Structure of Management Information

  An agent's management information is structured as a collection of managed objects that are stored in a virtual database called the MIB. Each object variable is identified using an object identifier (OID).

- Protocol Operations

  Describes the types of operations and messages that can be used to send management information between entities. Management objects are encapsulated within a protocol data unit (PDU).

- Administrative and Security Framework

  Defines the authentication, access control, and privacy mechanisms (if any) used to protect management information against unwanted operations.

Manager Entity

A manager is a node that actively participates in network management. It solicits and interprets data about network devices and network traffic, and typically interacts with a user to achieve the user's intentions. A manager can also initiate changes in an agent by changing the value of a variable on the agent node. Managers are frequently implemented as network management applications.

Agent Entity

An SNMP agent is software that resides on a network node and is responsible for communicating with managers regarding that node. The node is represented as a managed object, with various fields or variables, that are defined in the appropriate MIB. The agent has two purposes:

- To respond to requests from managers, supplying or changing the values of the object's variables as requested
- To generate traps to alert managers of events, such as a component failure, occurring at the node

Not all devices support SNMP directly. A device that does not directly support SNMP is called *foreign*. A *proxy agent* is an agent that serves a foreign device by translating between SNMP and the foreign device's protocol.

Dual Role Entity

A dual role entity is a node that has the ability to act both as a manager and an agent. Such nodes are often referred to as Mid-Level Managers (MLM). These types of entities solicit and interpret data about local network devices (acting as a manager). When this entity detects a predefined event, it sends an asynchronous notification to another manager (thus acting in an agent role). Dual role entities are usually used to localize network management traffic and distribute the network management workload among several nodes.

Management Information Base

This section reviews highlights of data representation and the concept of a MIB.

The MIB is a method of describing managed objects by specifying the names, types, and order of the fields, or variables, that make up the object. A MIB contains the definitions for a collection of standardized and non-standardized (vendor and experimental) objects.

The Internet MIB-II is one of many standard MIBs. MIB-II defines common objects for managing TCP/IP networks. Other standard MIBs manage specific network elements as well.

Note: For more information about these topics, refer to RFC 1155, RFC 1212, and RFC 1213.

MIB-II (RFC 1213) defines standardized objects for TCP/IP agents. To access the value of a MIB-II object, an SNMP manager sends a request to the agent representing the desired instance of the object. The request message contains MIB information (an OID) that lets the agent identify the specific objects. The corresponding response message from the agent carries the same identifying information.

OIDs

For the purpose of developing SNMP applications, an OID is a data type that precisely identifies a MIB-II object. An OID (sometimes referred to as the *registration ID*) consists of a sequence of nonnegative integers that describe a path through the object-naming hierarchy to the object. The naming hierarchy is commonly called the *naming tree*.

Naming Tree: The naming tree has the structure of a conventional tree with arbitrary breadth and depth. The nodes are labeled with nonnegative integers (each node among siblings must have a unique label).

Various organizations have administrative authority for assigning labels within subtrees of the naming tree. They can assign subordinate, or *child*, nodes and/or delegate this responsibility to other organizations. The root node of the naming tree has three children:

`ccitt(0)`	The administration authority for this branch is the International Telegraph and Telephone Consultative Committee (CCITT).
`iso(1)`	The administration authority for this branch is the International Organization for Standards, and the International Electrotechnical Committee (ISO/IEC). This is the path under which networking management is defined.
`joint-iso-ccitt(2)`	The administration authority for this branch is shared between CCITT and ISO/IEC.

Ultimately, every path through the naming tree terminates at a *leaf node*. The sequence of labels along the path (starting at the root) is the OID for the object named at the leaf.

OIDs in Practice: The convention for writing OIDs is called *dotted decimal notation*. An OID in dotted decimal notation consists of the integers of the OID in sequence with a dot between them. The prefix for the OIDs in the MIB-II is:

```
1.3.6.1.2.1
```

In the next example, the full name of the path is shown beneath the corresponding numerical identifiers in the OID:

```
1  . 3 . 6 . 1      .  2 . 1   . 6 . 7
iso.org.dod.internet.mgmt.mib-2.tcp.tcpAttemptFails
```

Similarly, the prefix for the OIDs in IBM's enterprise-specific MIBs is:

```
1.3.6.1.4.1.6.3
```

Extended MIBs

Many agents support extended MIBs, that define objects that are not included in standard MIBs. Your application can query an object from an extended MIB exactly as it would query a MIB-II object. Users should work with the proper registration authorities when defining MIB extensions.

Data Representation

Information is exchanged between SNMP entities using the basic encoding rules (BERs) defined for the abstract syntax notation (ASN.1). ASN.1 is a very rich data description language. The WinSNMP API takes care of the details of ASN.1 encoding and decoding; you do not have to deal directly with ASN.1 or the BER.

Highlights of SNMPv1

This section briefly discusses the protocol operations, data syntax, and security framework which comprise SNMP version 1.

SNMPv1 Protocol Operations

The following table describes the 5 basic operations (PDU types) found in SNMPv1.

PDU Type	Description
GetRequest	Used to request the value of one or more object variables managed by an agent.
GetNextRequest	Used to request the value of the object variables which immediately follow those specified in the request.
GetResponse	Used to carry response data to a GetRequest or GetNextRequest.
SetRequest	Used to write a new value into one or more object variables managed by an agent.
Trap	An unsolicited notification sent from an SNMP agent to an SNMP manager.

SNMPv1 Data Syntax

The following list identifies the few simple ASN.1 data types that are used in SNMPv1.

Type	Description
Integer	A simple type consisting of positive and negative whole numbers, including zero, and of arbitrary size up to 32 bits. Some objects restrict integer to a range.

Octet String	A simple type taking zero or more octets, each octet being an ordered sequence of 8 bits. The value of any octet in the string is unrestricted.
Object Identifier	An array of integers (unsigned longs) Each integer represents one element of the OID.
Counter	A nonnegative integer that calculates change and increases until it reaches a maximum value, then wraps around and starts increasing again from zero.
Gauge	A type representing a nonnegative integer, which may increase or decrease, but which latches at a maximum value.
TimeTicks	A type representing a nonnegative integer that counts the time in hundredths of a second since some event occurred.
IPaddress	A type representing a 32-bit Internet address. It is represented as an octet string of length 4, in network byte-order. When this ASN.1 type is encoded using the ASN.1 basic encoding rules, only the primitive encoding form is used.
Opaque	A type representing an arbitrarily-coded ASN.1 string, which has been coded into an octet string using the BERs.

SNMPv1 Administrative and Security Framework

The SNMPv1 framework associates each SNMP message with a community. An agent may exercise control of its management information by limiting the parts of its MIB tree which can be accessed (MIB view), along with the types of operations which can be performed, through the use of a community identifier.

Whenever a manager entity sends an SNMPv1 message to an agent entity, the message must include the correct community identifier in order to be allowed access to the desired data. An Agent may allow global access to its data by accepting a standardized community value of public, or may limit access by using a private value.

SNMP manager entities typically maintain a database of private community values for some or all of the agents with which it communicates.

Highlights of Community Based SNMPv2 (SNMPv2C)

Community based SNMPv2 (SNMPv2C) is the long awaited follow on to the SNMPv1 protocol and has been approved as a draft-standard by the Internet Engineering Steering Group (IESG) in January 1996. Improvements to SNMP include a larger set of PDU error status codes to describe failures, enhancements to the protocol operations, and an increased subset of ASN.1 syntax types for greater flexibility when defining new MIBs. SNMPv2C is defined in RFC 1901 through RFC 1908.

Noticeably absent in SNMPv2C are improvements to the security mechanisms for protecting management information from unwanted manipulation and disclosure as well as the ability to remotely configure security information at the Agent. One of the flaws in version 1 of SNMP was that only a clear text “community” string is used to prevent unauthorized agent operations. All management data (including the community string itself) flows unencrypted between a Manager and an Agent.

Standardizing a security framework within the IESG has proven to be problematic. The IESG decided to move the protocol improvements (mentioned above) forward but to continue using the “community based” security framework from SNMPv1 until consensus is reached on a new security solution.

SNMPv2C agents are currently defined to WinSNMP via the configuration file: /usr/OV/conf/snmpv2.conf.

SNMPv2C Protocol Operations

This section describes the enhanced list of protocol operations (PDU types) defined for SNMPv2C.

PDU Type	Description
GetRequest	Used to request the value of one or more object variables managed by an agent.
GetNextRequest	Used to request the value of the object variables which immediately follow those specified in the request.
Response	Used to carry data variables in response to a GetRequest, GetNextRequest, GetBulkRequest, or InformRequest PDU.
SetRequest	Used to write a new value into one or more object variables managed by an agent.
TrapRequest	This SNMPv1 data type is now obsolete.
GetBulkRequest	Used to request the values of a potentially large number of object variables. It replaces the use of multiple GetNextRequest PDUs when the efficient and rapid retrieval of large MIB tables is desired.
InformRequest	Used to send management information from one manager entity (or MLM) to another manager entity. After the receiving manager entity receives an InformRequest, it generates a response to the originating entity signifying receipt of the data.
SNMPv2Trap	An unsolicited notification sent from an agent to an SNMP manager.
Report	Semantics not presently defined. For use with future administrative and security frameworks.

SNMPv2C Data Syntax

This section describes the enhanced list of ASN.1 data types used in SNMPv2C.

Integer	Unchanged from SNMPv1. A simple type consisting of positive and negative whole numbers which can be represented using 32 bits.
Octet String	Unchanged from SNMPv1. A simple type taking zero or more 8-bit octets, each with a range 0 to 255.
Object Identifier	Unchanged from SNMPv1. A simple type which is an array of integers representing a MIB OID.

Address	Unchanged from SNMPv1. A simple type which is an octet string representing an IP address in network byte order.
TimeTicks	Unchanged from SNMPv1. A nonnegative integer type which counts the time in hundredths of a second since some event occurred.
Opaque	Used for backward-compatibility only. An octet string type used to represent an arbitrarily coded ASN.1 string.
Integer32	Indistinguishable from type Integer, but never needs more than 32 bits for a two's complement representation.
Counter32	Redefines the SNMPv1 Counter type as a 32-bit nonnegative increasing integer counter. When its maximum value is reached, it wraps around and begins increasing again from 0.
Gauge32	Redefines the SNMPv1 Gauge type as a 32-bit nonnegative integer which may increase or decrease, but will latch at its maximum value.
Counter64	A new 64-bit data type to be used only when the use of Counter32 type would wrap in less than one hour.
Unsigned32	A new nonnegative 32-bit type which is the nonnegative version of Integer32.

SNMPv2C Administrative and Security Framework

The community based administrative and security framework for SNMPv2C remain unchanged from SNMPv1.

Highlights of Secure User Based SNMPv2 (SNMPv2USEC)

In the absence of an IESG standard for secure SNMP, WinSNMP is employing the User-Based Security Model for SNMPv2 (SNMPv2USEC) to perform authenticated transactions with agents who also support SNMPv2USEC. SNMPv2USEC is one of the leading experimental models under consideration for future IESG standardization. The WinSNMP API supports secure SNMPv2USEC in its “transparent” mode only. SNMPv2USEC agent information is currently not exposed via the application interface, but only the local configuration file:
/usr/OV/conf/snmpv2.conf.

This allows WinSNMP applications to automatically obtain the benefits of SNMPv2 security now, while being insulated from future API changes which may be required to accommodate the eventual SNMPv2 security standard! Of course, NetView for AIX WinSNMP will fully support whatever model is finally selected as the IESG standard.

SNMPv2USEC Protocol Operations

The protocol operations (PDU types) for SNMPv2USEC remain unchanged from SNMPv2C with the exception of the Report PDU whose semantics have now been defined. The use of the Report PDU is reserved for SNMP internals and therefore outside the scope of this information.

SNMPv2USEC Data Syntax

The protocol operations (PDU types) for SNMPv2USEC remain unchanged from SNMPv2C.

SNMPv2USEC Administrative and Security Framework

The security framework for Secure SNMPv2 is designed to preserve the following properties:

Data Authentication	Describes the ability to ensure that an SNMP message has not been altered in transit by an unauthorized entity. It also provides a mechanism to verify the source identity of the message originator.
Data Privacy	Describes the ability to protect the information within an SNMP message from disclosure while in transit through encryption. **Note:** Due to US export regulations, NetView for AIX WinSNMP supports data authentication but *does not* support data privacy at this time.

SNMPv2USEC provides data authentication using the MD5 message digest algorithm. In this model, both the source and destination of a message have knowledge of a shared secret authentication key. When a manager wishes to send an authenticated request to an agent, it places the secret key or password (known by both entities) along with the management data into an SNMPv2 message. The entire message is then processed by the MD5 algorithm creating a digest (fingerprint). The secret key in the message is then replaced with this fingerprint, and the resulting message is sent.

Note: The management data within the message is not protected from disclosure; it is transmitted unencrypted along with its authentication fingerprint.

Upon receipt of the message, the agent replaces the incoming fingerprint with its value of the secret key and runs the message through the MD5 algorithm. A match between the resulting fingerprint and the incoming fingerprint indicates that the data portion of the message has not been altered from the data originally sent and that the originator had to have known the secret key. The message is therefore considered authentic. A fingerprint mismatch would have indicated that either the originator did not know the correct key or that the data portion of the message had been altered during transit.

Data privacy is provided by SNMPv2USEC using the Symmetric Encryption Protocol (DES). Both the source and destination SNMP entities have knowledge of a shared secret privacy key or password (which may be different from the Authentication key described earlier). When a manager wants to send a private request to an agent, it encrypts the entire authenticated message (both data and fingerprint portions) using DES and the privacy key. The resulting encrypted message is then transmitted to the destination.

Chapter 16. Programming with NetView for AIX WinSNMP

This chapter outlines some of the high level considerations relevant to the programming "model" envisioned by the NetView for AIX implementation of the *Windows SNMP Manager API Specification* (from here on referred to as WinSNMP). This model is meant to add background and context for evaluating the WinSNMP API. In general, although it is not possible to avoid all references to implementation details, WinSNMP tries to openly state all relevant implementation assumptions. For further information about WinSNMP, see Chapter 15, "Introduction to the NetView for AIX WinSNMP API" on page 237.

The major aspects of WinSNMP implementation that affect application development include:

- Levels of SNMP Support
- Transport Interface Support
- Entity/Context Translation Modes
- Local Database Information
- Session Characteristics
- Memory Management
- Asynchronous Model
- Polling and Retransmission
- Error Handling
- Data Types

Levels of SNMP Support

WinSNMP allows for multiple levels of SNMP support—explicitly for implementations and implicitly for applications.

These levels of support are independent of and unrelated to the modes of interpretation of entity and context arguments.

Note: The implementation will report its maximum level of SNMP support in response to the SnmpStartup function.

Implementations

The WinSNMP API allows an implementation to support any of four overlapping levels of SNMP operations:

- Level 0 = Message encoding/decoding only
- Level 1 = Level 0 + interaction with SNMPv1 agents
- Level 2 = Level 1 + interaction with SNMPv2 agents
- Level 3 = Level 2 + interaction with other SNMPv2 managers

Note: The NetView for AIX WinSNMP API provides Level 3 implementation.

Level 0 Implementations

Level 0 implementations must support all WinSNMP functions except those that require communication with other SNMP entities, namely:

- SnmpSendMsg
- SnmpRecvMsg
- SnmpRegister

Level 0 implementations exist to provide SNMP message encoding and decoding services to applications that do not require the communications transport services of the WinSNMP implementation, but still require WinSNMP services, such as:

- Local Database Functions
- SnmpEncodeMsg
- SnmpDecodeMsg

All WinSNMP implementations must include full Level 0 support.

Level 1 Implementations

Level 1 implementations support communications with SNMPv1 agents, in addition to providing full Level 0 support.

Because WinSNMP applications are structured to support SNMPv2, Level 1 implementations must support the requisite transformations specified in RFC 1908. (See "SNMP Information" on page 383 for a listing of RFCs.) For example, if a WinSNMP application submits a GetBulkRequest PDU to a Level 1 implementation, the WinSNMP implementation transforms the PDU into a GetNextRequest PDU, per RFC 1908, and proceeds accordingly.

WinSNMP always returns traps in SNMPv2 format, whether the trap emanates from an SNMPv1 agent or, as a notification, from an SNMPv2 agent. This behavior is also defined by RFC 1908.

Level 1 implementations must support the use of target agent addresses and community strings; but are not required to support any SNMPv2 mechanisms, other than the coexistence transformations mentioned above.

Level 2 Implementations

Level 2 implementations support communications with SNMPv2C agents, in addition to providing full Level 1 and Level 0 support.

Level 3 Implementations

Level 3 implementations support communications with other SNMPv2C management entities via the InformRequest PDU type, in addition to providing full Level 2, Level 1, and Level 0 support.

Applications

The WinSNMP API is oriented toward the writing of applications that are SNMPv2-enabled, at least in terms of their structure. A WinSNMP application may always use the relevant PDU types defined for SNMPv2 (as specified in the "Declarations" section of the /usr/OV/include/NVAIXwinsnmp.h file. This ensures that the implementation will perform the necessary transformations, in accordance with RFC 1908, when communicating with an SNMPv1 agent on behalf of the application. Likewise, a WinSNMP application always receives trap PDUs (via SnmpRecvMsg from the implementation) as SNMPv2 traps, even when the issuing entity is an SNMPv1 agent.

Note: It is possible for WinSNMP applications to utilize the implementation merely for SNMP message encoding and decoding and to bypass the WinSNMP implementation with respect to communications with the destination entities. In this mode, the application must perform the necessary GetResponse and trap PDU transformations for itself, at its own discretion.

Transport Interface Support

For everything above Level 0, the WinSNMP implementation conducts the communications transactions with the SNMP agents on behalf of the applications. Nothing in WinSNMP attempts to dictate how an implementation (or an application) will actually execute the communications process with remote entities.

A number of options exist. They are not necessarily mutually exclusive—several might be used by an implementation with one or more in the same or a different combination being used by its client applications.

Note: IBM WinSNMP supports the UDP and TCP transport protocols.

Entity/Context Translation Modes

WinSNMP applications have the capability of instructing the implementation to interpret entity/context arguments as either a literal SNMPv1 agent address and community string, respectively, or as a SNMPv2C agent address and community string. An alternative to either of these modes is that in which these arguments are interpreted as user or application "friendly" names for entities and managed object collections to be translated into their respective SNMPv1, SNMPv2C, or SNMPv2USEC components via the implementation's local database.

The three entity and context translation modes are:

SNMPAPI_TRANSLATED	Translate via Local Database look-up
SNMPAPI_UNTRANSLATED_V1	Literal transport address and community string for an SNMPv1 agent
SNMPAPI_UNTRANSLATED_V2	Literal transport address and community string for an SNMPv2C agent

Important

SNMPv2USEC entities may be accessed via SNMPAPI_TRANSLATED mode *only*. Specific security information for these agents are only accessible though a database look-up. This is due to that fact that the SNMPv2 User Based Security (SNMPv2USEC) model is currently not an Internet standard. Not exposing SNMPv2USEC details to the application allows for a seamless transition to whatever the final Internet SNMP Security standard happens to be.

The WinSNMP implementation always identifies its current default entity/context translation mode setting in the return value from the SnmpStartup function (which may be called multiple times). A WinSNMP application may request a different entity/context translation mode at any time with the SnmpSetTranslateMode function.

The sample code which follows includes literal string representations of some of the arguments to the WinSNMP functions. This is merely for expository purposes. In the interests of internationalization and localization application writers are encouraged to isolate all such text string values in StringTables in separate resource files or to use some similar technique to modularize such strings out of the operating logic of their applications.

Context string arguments are passed as octet string structures (smiOCTETS descriptors) since both SNMPv1 and SNMPv2C community strings can contain any values, not just those from the ASCII or DisplayString character set.

SNMPAPI_TRANSLATED Mode

When the translation mode is set to SNMPAPI_TRANSLATED, an application makes calls similar to the following:

```
LPCSTR entityName = "Secure_USEC_Hub";
smiOCTETS contextName;
contextName.ptr = "Secure_USEC_Hub";
contextName.len = lstrlen(contextName.ptr);
hAgent = SnmpStrToEntity(hSomeSession, entityName);
hView  = SnmpStrToContext(hSomeSession, const &contextName);
```

The implementation looks up "Secure_USEC_Hub" in its local database (currently the /usr/OV/conf/snmpv2.conf file) and if successful, assembles the appropriate internal data structures (including any SNMPv2USEC security data) and returns HANDLE values for use by the application.

SNMPAPI_UNTRANSLATED_V1 Mode

When the translation mode is set to SNMPAPI_UNTRANSLATED_V1, an application makes calls similar to the following:

```
LPCSTR entityName = "192.151.207.34";
smiOCTETS contextName;
contextName.ptr = "public";
contextName.len = lstrlen (contextName.ptr);
hAgent = SnmpStrToEntity (hSomeSessin, entityName);
hView   = SnmpStrToContext (hSomeSession, const &contextName);
```

The implementation assumes, based on the SNMPAPI_UNTRANSLATED_V1 setting for hSomeSession, that "192.151.207.34" equates to an IP address to be reached via UDP port 161, and that this value is being passed as a far pointer to a constant NULL terminated text string (LPCSTR) that it must first convert to dotted decimal notation. The setting also indicates that an SNMPv1 packet should be constructed when communicating with this entity.

SNMPAPI_UNTRANSLATED_V2 Mode

When the translation mode is set to SNMPAPI_UNTRANSLATED_V2, an application makes calls similar to the following:

```
LPCSTR entityName = "192.151.207.34";
smiOCTETS contextName;
contextName.ptr = "public";
contextName.len = lstrlen (contextName.ptr);
hAgent = SnmpStrToEntity (hSomeSessin, entityName);
hView   = SnmpStrToContext (hSomeSession, const &contextName);
```

The implementation assumes, based on the SNMPAPI_UNTRANSLATED_V1 setting for hSomeSession, that "192.151.207.34" equates to an IP address to be reached via UDP port 161, and that this value is being passed as a far pointer to a constant NULL terminated text string (LPCSTR) that it must first convert to dotted decimal notation. The setting also indicates that an SNMPv2C packet should be constructed when communicating with this entity.

Local Database

WinSNMP *must* meet at least the following four objectives:

1. A WinSNMP application must have full access to all components of the SNMP message issued by the WinSNMP implementation. At the extreme, the SnmpEncodeMsg and SnmpDecodeMsg functions enable access to and manipulation of fully-serialized, ready-for-transport SNMP messages.
2. A WinSNMP application must not have to incorporate WinSNMP implementation-specific routines or data structures to utilize any of the functionality defined by WinSNMP itself. Each WinSNMP implementation may use private mechanisms external to the WinSNMP applications, but all necessary interfaces to these mechanisms will be via the defined WinSNMP APIs only.
3. A WinSNMP application must not have to know the SNMP version level of the target SNMP entities acting in an agent role. The WinSNMP implementation performs all necessary mappings between SNMPv1, SNMPv2C, and SNMPv2USEC in accordance with the appropriate RFCs, and especially RFC 1908. With respect to agent addressing, this is especially true for TRANSLATED mode access; for protocol operations it holds regardless of the entity/context translation mode in effect.
4. One implication of the foregoing requirement is that the SNMPv1 message format must fit neatly within the structure adopted for the SNMPv2 message format. This statement applies to WinSNMP messages only—it is not meant in any way to limit or modify anything in RFC 1908.

Sessions

The session created by the SnmpCreateSession function is used to manage the link between the WinSNMP application and the WinSNMP interface implementation. That is, the session is the unit of resource and communications management between a calling WinSNMP application and its supporting WinSNMP implementation. A well-behaved WinSNMP application uses the session construct to logically organize its operations and to minimize resource requirements on the implementation. The following statements summarize the role and certain attributes of WinSNMP sessions:

- A session is opened with SnmpCreateSession, and closed with SnmpClose.
- A session-id is returned by the SnmpCreateSession function to the application as a HANDLE variable, which the implementation may use internally to manage resources.
- The minimum number of concurrent sessions which an implementation must support is one.
- The maximum number of concurrent sessions is undefined and is implementation-specific and, possibly, resource-dependent.

  When an application's request to open a session cannot be granted because of the limitations stated above, the implementation returns SNMPAPI_FAILURE to

SnmpCreateSession and sets SnmpGetLastError or SnmpGetLastErrorStr to report SNMPAPI_ALLOC_ERROR.

- All WinSNMP API functions, except for SnmpCreateSession, which return HANDLE variables include a session-id handle as an input parameter. The implementation uses the session-id handle internally to manage and account for resources on behalf of the session.
- HANDLE variables created under one open session can be utilized by other open sessions (if any) within a given application (task). Optionally, an implementation may internally share HANDLE variables among sessions in separate applications.

  Note: This optional resource efficiency, if it is supported by an implementation, is totally transparent to the application.
- When an application closes a session by executing the SnmpClose function, all resources created on behalf of that session by the implementation, and not previously freed by the application, are freed automatically by the implementation. If an implementation supports the optional sharing of HANDLE variables among open sessions across multiple applications, then the resources are not physically freed until the final open session that created the resources closes.
- Sessions may have other attributes, above and beyond those discussed above (for example, the dstEntity and context interpretation modes of TRANSLATED, UNTRANSLATED_V1, and UNTRANSLATED_V2).

Memory Management

The allocation, ownership, deallocation, and garbage collection of memory objects is often a troublesome issue in a complex multiprovider programming arrangement. This issue can be resolved with an understanding of the options and the rules, agreeing to a division of labor, authority, and responsibility among the components; and competence and diligence in implementing such an agreement.

In WinSNMP programming, it is important to remember that the implementation is actually just an extension of the calling application. Applications can allocate, use, and deallocate memory; if they terminate without freeing allocated memory, the operating system deallocates it for them automatically.

The WinSNMP arrangement includes three different kinds of memory objects:

- HANDLE'd resources
- C-style (NULL terminated) strings
- Non-scalar WinSNMP API data types of variable length

HANDLE'd Resources

There are five varieties of HANDLE'd resources:

- Sessions
- Entities
- Contexts
- Protocol data units (PDUs)
- VarBindLists (VBLs)

These objects are accessed by way of handles for two reasons:

- To hide their structures from the applications; and
- To permit implementations to optimize or differentiate themselves as compared with their construction and manipulation of these objects "behind" the API.

All HANDLE'd objects are of data type HSNMP_<object_tag> and are always owned by the implementation. An application may request their creation and may signal their eligibility for deletion and reclamation, but these operations (like all others concerning these objects) are indirect...the realization is up to the implementation.

C-Style Strings

The C-style (NULL terminated) strings are provided mainly for convenience to easily convert entity and OID objects to and from the most common string representation. The WinSNMP functions that use C-style strings are limited to: SnmpStrToEntity, SnmpEntityToStr, SnmpStrToOid, and SnmpOidToStr. (The inclusion of Str in the name is a bit misleading in the case of the SnmpStrToContext and SnmpContextToString functions, as the context parameter in these functions must be an SNMP-style octet string to accommodate the legal data values.)

The application is entirely responsible for allocating, managing, and freeing this memory, as might be appropriate to its specific operating requirements or circumstances. This requires passing a size parameter to the implementation in functions which use pointers to C-style string variables as output arguments (for example, SnmpEntityToStr and SnmpOidToStr).

Descriptors

These are the three non-scalar WinSNMP API data types, of variable length:

- smiOCTETS
- smiOID
- smiVALUE

All three are structures. The first two are both descriptor structures, consisting of two members: len and ptr. For smiOCTETS, len is an unsigned long integer (smiUINT32) value indicating the number of bytes in the subject octet string (no necessary NULL terminating byte) and ptr is a far pointer to a byte array containing the octet string. For smiOID, len is an unsigned long integer value indicating the number of unsigned long integers in the subject OID and ptr is a far pointer to an array of unsigned long integers representing the OID's sub-identifiers.

The smiVALUE structure also consists of two members, but is different and a bit more complex. The first member is an unsigned long integer indicating the syntax of the second member. The second member is the union of all the possible WinSNMP API data types. A calling application must first check the syntax member of a returned smiVALUE structure to know how to translate the second member, which might be a simple scalar value or might be one of the WinSNMP API structures with defined syntax (including an smiOCTETS, or one of its derivatives such as smiIPADDR, or an smiOID). In actuality, the smiVALUE structure is not a problem—it is always of a fixed size.

It is only when its syntax member indicates that the value member is either an smiOCTETS or an smiOID structure (which contain pointers to variable length data) that the memory management agreement becomes important. It is important to know who assigns the pointers (for example, allocates the memory), who fills in the len members, who owns these objects, and who is responsible for freeing the resources when they are no longer needed or in cases of memory shortage.

Fortunately, the statement of this problem is more complex than the statement of its resolution!

- For input parameters, the application provides the structure and populates its members (for example, allocates the memory for the variable length objects).
- For output parameters, the application again provides the structure, but the implementation populates its members (for example, allocates the memory for the variable length objects).
- The application must use an appropriate function (for example, GlobalFreePtr) to free the memory that it has allocated for such input parameters and must use the SnmpFreeDescriptor WinSNMP function to free the memory allocated by the implementation for these output parameters.

The combined effects of this particular agreement yield substantial benefits:

- It clearly delineates a small number of cooperative memory management requirements.
- It clearly assigns responsibility in each case.
- It reduces the likelihood of over allocation of temporary buffer space.
- It reduces the likelihood of unnecessary buffer copying (from maximum size temporary buffers to the right size working buffers).
- It leverages a "natural" memory management posture while providing independent flexibility in this area to both applications and implementations alike.

Summary

As a general rule, the WinSNMP application is responsible for freeing all WinSNMP resources allocated through calls to the WinSNMP API using the following functions:

SnmpFree<*xxx*>	Entity, Context, PDU, VBL, Descriptor
SnmpClose	Session
SnmpCleanup	Task

These calls are cumulative, in the sense that SnmpFree<*xxx*> frees a single specific HANDLE'd resource, SnmpClose frees all such resources allocated to a given session within the calling task, and the session HANDLE'd resource itself, SnmpCleanup, performs (in effect) an SnmpClose on all open sessions within the task. Applications are encouraged to use these in the order shown as appropriate to the application's processing logic.

Important: Every time a WinSNMP function call results in the return of a HANDLE'd object resource to the application, each and every such resource will be a new resource. In this context, new means a unique value for that kind of resource at that instant in the calling application. This means, for example, that it is safe—with respect to a subsequent SnmpRecvMsg call—to free the srcEntity, dstEntity, context, and PDU resources right after calling SnmpSendMsg. Note that a given application for example, might need or want to retain them, of course. If you built these resources for a polling operation, you will probably want to retain them for the next iteration...and in many cases you will want to match up out-bound RequestIDs with received RequestIDs.

An allocated HANDLE'd object resource is never freed by the implementation except upon request by the application using one of the three types of calls outlined at the top of this section or—as an optional capability of an implementation—upon abnormal termination of an application which otherwise left resources allocated. It is the application's responsibility to request the creation of and the deletion of all WinSNMP HANDLE'd object resources. The implementation is free to perform these operations any way it wants internally, as long as the external appearance to the application accords with this set of specifications.

Asynchronous Model

One contemporary programming model has applications "driven" by the receipt and processing of asynchronous message-events. This asynchronous callback-driven model maps well to modern object-oriented theory, the SNMP distributed management paradigm, and the X-Windows programming and runtime environments. Likewise, although WinSNMP does not presume any particular transport mechanism for the conveyance of SNMP messages between managers and agents, SNMP is a datagram-based protocol, in which no actual channel (virtual circuit) is established between remote entities. This behavior also maps well to the message-driven programming model. For those reasons, among others, this is the programming model adopted by WinSNMP.

Modern callback-driven applications typically must respond to other kinds of important events, some of which may rely on synchronous relationships. Actually, all of the functions specified in the WinSNMP API have a synchronous component—most are totally synchronous; these three critical functions have an asynchronous dimension:

- SnmpSendMsg
- SnmpRecvMsg
- SnmpRegister

Of these, SnmpRecvMsg has the most impact on asynchronous operations.

The basic asynchronous model for programming with WinSNMP follows these steps:

1. The application opens a session with the WinSNMP implementation (with the SnmpCreateSession function).
2. If the application is interested in receiving traps, it indicates this (with the SnmpRegister function).
3. The application prepares one or more PDUs for transmission to and processing by the WinSNMP implementation via WinSNMP messages (using SnmpCreatePdu and other PDU, variable-binding, and utility functions).
4. The application submits one or more asynchronous requests consisting of SNMP PDU and message "wrapper" elements (with the SnmpSendMsg function).

5. The application receives notification that a response to a request is available or that a registered trap has occurred (via the callback function specified in the SnmpCreateSession function).
6. The application retrieves the response (with the SnmpRecvMsg function).
7. The application processes the response as appropriate (using application-specific logic).
8. The application closes the WinSNMP session (with the SnmpClose function).

In general, steps 2 on page 255 through 7 can take place in nearly any order and at any time during program execution.

Polling and Retransmission

Given the asynchronous nature of both SNMP itself and the WinSNMP SnmpSendMsg, SnmpRecvMsg, and SnmpRegister functions, users of this API (for example, implementors and applications writers) must be concerned with timeout and retry issues. Taken together, timeout and retry will be referred to hereinafter as retransmission.

Note: Back-off mechanisms are not currently included.

Applications have sole responsibility for polling: for example, they establish the frequency, and initiate transactions and timer management. This ensures that applications have knowledge of the request-id component of the out-going PDUs.

With respect to retransmission, applications clearly have the primary responsibility, regarding both policy and execution. Implementations must provide retransmission policy support (via their local database) and may optionally provide retransmission execution support.

Accordingly, in WinSNMP applications the timeout period, in practice, refers to the elapsed time between an application's issuance of an SnmpSendMsg request and receipt of the corresponding message via the SnmpRecvMsg function. From the perspective of the implementation, the timeout period refers to the elapsed time between the actual sending of an SNMP request message to a destination entity and the receipt of the SNMP response message from that destination.

The fundamental retransmission policy mechanism will be the local database. Each potential destination entity entry in the local database includes the timeout (elapsed time in seconds) and retry (count) elements among others. These values can be stored in and retrieved from the local database by an application with the Snmp[Get/Set]Timeout and Snmp[Get/Set]Retry functions. At runtime, an application may elect to use, update, or ignore the default values in the local database. When an implementation that supports retransmit execution is operating in retransmit mode, it must use the timeout and retry values from the local database for the respective destination entities.

None of the foregoing precludes or impedes the out-of-the-box mode of operation. An implementation should boot up with some generic default values in its (conceptual) local database for use when an application initializes entities in the SNMPAPI_UNTRANSLATED_V1 or SNMPAPI_UNTRANSLATED_V2 mode.

So, for WinSNMP, the following summarizes the timeout and retry elements approach:

- The application manages the policy via the local database functions by storing desired values for each destination entity. Optionally, the implementation may also update the actual observed values in its local database for subsequent use by the application in adjusting the desired (policy) values.
- The application executes the policy, at its discretion. That is, when it issues a request (via SnmpSendMsg) and wants to monitor the timeout event, it sets a timer (most likely using the desired timeout value retrieved from the local database).
- If the response comes in before the timer goes off, it cancels the timer. If the timer expires, the application decides whether to retry (most likely, but not only, based on the retry count value retrieved from the local database).
- If, during the course of execution, the application determines that either the default timeout or retry values are inappropriate, it can either ignore that fact, or change its runtime behavior accordingly, or modify the default values for the respective entities in the local database.
- Certain network smart applications might populate and update the default values in the local database, while many more network agnostic applications just use the default values, whether just for its policy (when the implementation actually does the execution) or for both policy and execution purposes.
- Applications may request that the implementation execute the retransmission policy (using the values in the local database) via the SnmpSetRetransmitMode function, with (SNMPAPI_ON). A valid response to this request by a compliant implementation is either SNMPAPI_SUCCESS or SNMPAPI_MODE_INVALID.
- The application may elect to leave retransmission execution entirely to the implementation or to augment it with its own execution. An application can use SnmpSetRetransmit again, with (SNMPAPI_OFF), to turn off the implementation in this regard.
- When the implementation executes the retransmission policy, it repeats the original request-id component in each retransmitted PDU.
- When the implementation responds to the SnmpSetRetransmitMode (SNMPAPI_ON) request with the SNMPAPI_MODE_INVALID error, the application must assume all responsibility for execution of the retransmission policy.

RequestIDs

This WinSNMP v1.1a release states that:

- RequestID of 0 must be allowed in PDUs.
- WinSNMP API functions cannot use RequestID as a return value, since SNMPAPI_FAILURE == 0.
- SnmpSendMsg and SnmpRecvMsg return either SNMPAPI_SUCCESS [1] or SNMPAPI_FAILURE [0], not the RequestID.
- Applications can use the SnmpGetPduData function to retrieve a PDU's RequestID when necessary.
- An application can ask the implementation to generate and assign a RequestID to a PDU by passing zero in the RequestID parameter of the SnmpCreatePdu

function and can then determine that value, if desired with the SnmpGetPduData function, per the previous bullet item.

- The implementation uses its best efforts to generate RequestIDs that are temporally unique internally (that is, a simple incrementing algorithm will suffice). It does not have to avoid conflicts with externally generated RequestIDs. It may assign zero as a generated RequestID in the normal course of events.
- An application that wants to force a RequestID of zero (or any other value) can use the SnmpSetPduData function to do so.

Error Handling

All WinSNMP functions have an immediate return value. If this value is SNMPAPI_FAILURE (0), it means that the implementation detected or encountered an error. The application must then call the SnmpGetLastError or SnmpGetLastErrorStr function to retrieve the extended error information that describes the specific problem encountered.

The distinction between SNMP error codes and SNMP API error codes in the context-specific section is more significant. The former are fixed by the RFCs; the latter are creations of WinSNMP.

Common Error Codes

Any WinSNMP function can fail with any one of the following error codes returned via SnmpGetLastError or SnmpGetLastErrorStr:

SNMPAPI_NOT_INITIALZED
SNMPAPI_ALLOC_ERROR
SNMPAPI_OTHER_ERROR

SNMPAPI_NOT_INITIALIZED signals that SnmpStartup was not successfully executed, either since program execution began or since SnmpCleanup successfully completed. If SnmpStartup fails, an immediate call to SnmpGetLastError or SnmpGetLastErrorStr (before any other WinSNMP calls) returns the error code applicable to the failure of SnmpStartup; all subsequent calls to WinSNMP functions before a successful SnmpStartup execution will fail with SNMPAPI_NOT_INITIALIZED.

SNMPAPI_ALLOC_ERROR signals that the implementation was unable to obtain sufficient resources to carry out the requested action. Applications should respond by freeing resources, or by reducing the resource requirements of the request, or by informing the user (for example, via MessageBox or log file entry) and facilitating a normal shutdown via SnmpClose calls or SnmpCleanup.

SNMPAPI_OTHER_ERROR signals that an unknown, undefined, or otherwise indeterminate error occurred. Implementations may provide an optional, ancillary, and independent means of providing additional feedback to the user for subsequent problem resolution. In most cases, applications should attempt to shutdown gracefully via SnmpClose calls or SnmpCleanup after receiving this error.

Context-Specific Error Codes

The following lists are excerpted from the "Declarations" section of the /usr/OV/include/WinSNMP.h file. They are included here mainly as a place-holder for a future elaboration of each error condition, similar to what was done in “Common Error Codes” on page 258.

```
/* Syntax Values for Exception Conditions in SNMPv2 Response Varbinds */
#define SNMP_VALUE_NOSUCHOBJECT    (ASN_CONTEXT | ASN_PRIMITIVE | 0x0)
#define SNMP_VALUE_NOSUCHINSTANCE  (ASN_CONTEXT | ASN_PRIMITIVE | 0x1)
#define SNMP_VALUE_ENDOFMIBVIEW    (ASN_CONTEXT | ASN_PRIMITIVE | 0x2)

/* SNMP Error Codes Returned in Error_status Field of PDU...Not API Error Codes */
/* Error Codes Common to SNMPv1 and SNMPv2 */
#define SNMP_ERROR_NOERROR                                     0
#define SNMP_ERROR_TOOBIG                                      1
#define SNMP_ERROR_NOSUCHNAME                                  2
#define SNMP_ERROR_BADVALUE                                    3
#define SNMP_ERROR_READONLY                                    4
#define SNMP_ERROR_GENERR                                      5
/* Error Codes Added for SNMPv2 */
#define SNMP_ERROR_NOACCESS                                    6
#define SNMP_ERROR_WRONGTYPE                                   7
#define SNMP_ERROR_WRONGLENGTH                                 8
#define SNMP_ERROR_WRONGENCODING                               9
#define SNMP_ERROR_WRONGVALUE                                 10
#define SNMP_ERROR_NOCREATION                                 11
#define SNMP_ERROR_INCONSISTENTVALUE                          12
#define SNMP_ERROR_RESOURCEUNAVAILABLE                        13
#define SNMP_ERROR_COMMITFAILED                               14
#define SNMP_ERROR_UNDOFAILED                                 15
#define SNMP_ERROR_AUTHORIZATIONERROR                         16
#define SNMP_ERROR_NOTWRITABLE                                17
#define SNMP_ERROR_INCONSISTENTNAME                           18

/* WinSNMP API Function Return Codes */
#define SNMPAPI_FAILURE          0     /* Generic error code */
#define SNMPAPI_SUCCESS          1     /* Generic success code */
/* WinSNMP API Error Codes (for SnmpGetLastError) */
#define SNMPAPI_ALLOC_ERROR      2     /* Error allocating memory */
#define SNMPAPI_CONTEXT_INVALID  3     /* Invalid context parameter */
#define SNMPAPI_CONTEXT_UNKNOWN  4     /* Unknown context parameter */
#define SNMPAPI_ENTITY_INVALID   5     /* Invalid entity parameter */
#define SNMPAPI_ENTITY_UNKNOWN   6     /* Unknown entity parameter */
#define SNMPAPI_INDEX_INVALID    7     /* Invalid VBL index parameter */
#define SNMPAPI_NOOP             8     /* No operation performed */
#define SNMPAPI_OID_INVALID      9     /* Invalid OID parameter */
#define SNMPAPI_OPERATION_INVALID 10   /* Invalid/unsupported operation */
#define SNMPAPI_OUTPUT_TRUNCATED 11    /* Insufficient output buf len */
#define SNMPAPI_PDU_INVALID      12    /* Invalid PDU parameter */
#define SNMPAPI_SESSION_INVALID  13    /* Invalid session parameter */
#define SNMPAPI_SYNTAX_INVALID   14    /* Invalid syntax in smiVALUE */
#define SNMPAPI_VBL_INVALID      15    /* Invalid VBL parameter */
#define SNMPAPI_MODE_INVALID     16    /* Invalid mode parameter */
#define SNMPAPI_SIZE_INVALID     17    /* Invalid size/length parameter */
#define SNMPAPI_NOT_INITIALIZED  18    /* SnmpStartup failed/not called */
#define SNMPAPI_MESSAGE_INVALID  19    /* Invalid SNMP message format */
#define SNMPAPI_HWND_INVALID     20    /* Invalid Window handle */
```

```
/* Others will be added as needed */
#define SNMPAPI_OTHER_ERROR       99    /* For internal/undefined errors */
```

Transport Error Reporting

In the case of errors that are detected at the time of accepting a request to send or to receive a packet, these are returned synchronously by SnmpSendMsg, SnmpRecvMsg, or SnmpRegister via a return code of SNMPAPI_FAILURE (which the application must follow-up with a call to SnmpGetLastError (to retrieve the extended error code). In the case of errors which are detected after the packet has gone out onto the wire, the WinSNMP implementation sends a packet receipt notification to the affected session and these errors are returned via an SNMPAPI_FAILURE indication from the next SnmpRecvMsg call on that session.

The generic transport layer (TL) error codes for WinSNMP are:

```
#define SNMPAPI_TL_NOT_INITIALIZED  100  /* Transport layer not initialized */
#define SNMPAPI_TL_NOT_SUPPORTED    101  /* Transport does not support protocol */
#define SNMPAPI_TL_NOT_AVAILABLE    102  /* Network subsystem has failed */
#define SNMPAPI_TL_RESOURCE_ERROR   103  /* Transport resource error */
#define SNMPAPI_TL_UNDELIVERABLE    104  /* Destination unreachable */
#define SNMPAPI_TL_SRC_INVALID      105  /* Source endpoint invalid */
#define SNMPAPI_TL_INVALID_PARAM    106  /* Input parameter invalid */
#define SNMPAPI_TL_IN_USE           107  /* Source endpoint in use already */
#define SNMPAPI_TL_TIMEOUT          108  /* No response within Timeout interval */
#define SNMPAPI_TL_TOO_BIG          109  /* PDU too big for send/receive */
#define SNMPAPI_TL_OTHER            199  /* Undefined transport error */
```

Specific transport layer errors are listed as appropriate in the definitions of the SnmpRegister, SnmpSendMsg, and SnmpRecvMsg functions in the *NetView for AIX Programmer's Reference*.

Implementations should attempt to map specific transport errors to one of the generic transport errors. If the mapping is not possible, the implementation should return SNMPAPI_TL_OTHER. This error is preferred over SNMPAPI_OTHER_ERROR, for unmapped transport layer errors.

WinSNMP Data Types

The following is an excerpt from the "Declarations" section of the /usr/OV/include/WinSNMP.h file.

```
/* WinSNMP API Type Definitions  */
typedef  HANDLE            HSNMP_SESSION,        FAR *LPHSNMP_SESSION;
typedef  HANDLE            HSNMP_ENTITY,         FAR *LPHSNMP_ENTITY;
typedef  HANDLE            HSNMP_CONTEXT,        FAR *LPHSNMP_CONTEXT;
typedef  HANDLE            HSNMP_PDU,            FAR *LPHSNMP_PDU;
typedef  HANDLE            HSNMP_VBL,            FAR *LPHSNMP_VBL;
typedef unsigned char      smiBYTE,              FAR *smiLPBYTE;
/* SNMP-related types from RFC1442 (SMI) */
typedef signed long        smiINT,               FAR *smiLPINT;
typedef smiINT             smiINT32,             FAR *smiLPINT32;
typedef unsigned long      smiUINT32,            FAR *smiLPUINT32;
typedef struct {
     smiUINT32 len;
     smiLPBYTE ptr;}       smiOCTETS,            FAR *smiLPOCTETS;
```

```
typedef const smiOCTETS                               FAR *smiLPCOCTETS;
typedef smiOCTETS          smiBITS,             FAR *smiLPBITS;
typedef struct {
     smiUINT32   len;
     smiLPUINT32 ptr;}      smiOID,              FAR *smiLPOID;
typedef const smiOID                            FAR *smiLPCOID;
typedef smiOCTETS          smiIPADDR,           FAR *smiLPIPADDR;
typedef smiUINT32          smiCNTR32,           FAR *smiLPCNTR32;
typedef smiUINT32          smiGAUGE32,          FAR *smiLPGAUGE32;
typedef smiUINT32          smiTIMETICKS,        FAR *smiLPTIMETICKS;
typedef smiOCTETS          smiOPAQUE,           FAR *smiLPOPAQUE;
typedef smiOCTETS          smiNSAPADDR,         FAR *smiLPNSAPADDR;
typedef struct {
        smiUINT32 hipart;
        smiUINT32 lopart;}  smiCNTR64,           FAR *smiLPCNTR64;

/* Structure used to compose a value member for a variable binding */
typedef struct {                  /* smiVALUE portion of VarBind */
        smiUINT32   syntax;       /* Insert SNMP_SYNTAX_<type> */
        union {
        smiINT      sNumber;      /* SNMP_SYNTAX_INT
                                     SNMP_SYNTAX_INT32 */
        smiUINT32   uNumber;      /* SNMP_SYNTAX_UINT32
                                     SNMP_SYNTAX_CNTR32
                                     SNMP_SYNTAX_GAUGE32
                                     SNMP_SYNTAX_TIMETICKS */
        smiCNTR64   hNumber;      /* SNMP_SYNTAX_CNTR64 */
        smiOCTETS   string;       /* SNMP_SYNTAX_OCTETS
                                     SNMP_SYNTAX_BITS
                                     SNMP_SYNTAX_OPAQUE
                                     SNMP_SYNTAX_IPADDR
                                     SNMP_SYNTAX_NSAPADDR */
        smiOID      oid;          /* SNMP_SYNTAX_OID */
        smiBYTE     empty;        /* SNMP_SYNTAX_NULL
                                     SNMP_SYNTAX_NOSUCHOBJECT
                                     SNMP_SYNTAX_NOSUCHINSTANCE
                                     SNMP_SYNTAX_ENDOFMIBVIEW */
            }       value;        /* union */
        }           smiVALUE,     FAR *smiLPVALUE;
typedef const smiVALUE            FAR *smiLPCVALUE;
```

Integers

The standard integer type used by WinSNMP is unsigned long (smiUINT32). In a few places, parameters are specified as signed long (smiINT) to comply with data elements defined in the respective RFCs. (This is especially true of some of the PDU components.)

Pointers

All pointer variables used in this specification are far pointers; large model programming is assumed.

Function Returns

All return values from WinSNMP functions fall into two categories:

- A HANDLE to a resource allocated by the implementation on behalf of the application, including:
  - Sessions (HSNMP_SESSION)
  - Entities (HSNMP_ENTITY)
  - Contexts (HSNMP_CONTEXT)
  - PDUs (HSNMP_PDU)
  - Variable Binding Lists (HSNMP_VBL)
- A long unsigned integer (smiUINT32) value representing a status (SNMPAPI_STATUS).
  - SNMPAPI_FAILURE (equates to 0 or NULL)
  - SNMPAPI_SUCCESS (equates to 1 or a positive count)

Descriptors

Two important WinSNMP data types, octet strings and OIDs, take the form of descriptors. A descriptor is a structure consisting of a length member (len) and a pointer member (ptr), of the appropriate type (for example, smiLPBYTE or smiLPUINT32, respectively), to the actual data item of interest. Either of these two descriptors can occur in the value member of an smiVALUE structure, as can any of the scalar WinSNMP types.

When a descriptor that has been allocated by the application is actually populated (has its len and ptr members defined for it) by the implementation, the application must eventually call the SnmpFreeDescriptor function to enable the implementation to release the resources associated with ptr member.

For more detailed memory management information, see “Descriptors” on page 253, in the Memory Management section.

WinSNMP Interfaces

This section comprises the function reference for WinSNMP. In general, not a lot of significance attaches to the categorization or ordering of these functions:

- Local database functions
- Communications functions
- Entity/context functions
- PDU functions
- Variable binding functions
- Utility functions

Local Database Functions

The functions in this section concern manipulation of the local database of SNMP administrative information.

The term database in this context is not meant to imply any particular data storage, access, or manipulation techniques. The WinSNMP implementation is the owner of the local database and may utilize any proprietary mechanisms it considers best, as long as all the functions defined in this section are fully supported and no additional implementation-specific functions are required of a WinSNMP application to utilize the local database. Compliant WinSNMP implementations may require additional implementation-specific mechanisms external to a WinSNMP application (for example, setting an environment variable to point to a local database file).

The functions in this section are:

Return Type	**Procedure Name**	**Parameters**
SNMPAPI_STATUS	SnmpGetTranslateMode	(OUT smiLPUINT32 nTranslateMode);
SNMPAPI_STATUS	SnmpSetTranslateMode	(IN smiUINT32 nTranslateMode);
SNMPAPI_STATUS	SnmpGetRetransmitMode	(OUT smiLPUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpSetRetransmitMode	(IN smiUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpGetTimeout	(IN HSNMP_ENTITY hEntity, OUT smiLPTIMETICKS nPolicyTimeout, OUT smiLPTIMETICKS nActualTimeout);
SNMPAPI_STATUS	SnmpSetTimeout	(IN HSNMP_ENTITY hEntity, IN smiTIMETICKS nPolicyTimeout);
SNMPAPI_STATUS	SnmpGetRetry	(IN HSNMP_ENTITY hEntity, OUT smiLPUINT32 nPolicyRetry, OUT smiLPUINT32 nActualRetry);
SNMPAPI_STATUS	SnmpSetRetry	(IN HSNMP_ENTITY hEntity, IN smiUINT32 nPolicyRetry);

Communications Functions

The functions in this section concern communications between the calling WinSNMP application and the serving WinSNMP implementation. Communications to and from other management entities, can reside on the local machine, on a connected LAN or WAN, or an internet. They are handled by the WinSNMP implementation on behalf of the WinSNMP application, and without any overt orchestration by the latter.

The functions in this section are:

Return Type	**Procedure Name**	**Parameters**
SNMPAPI_STATUS	SnmpStartup	(OUT smiLPUINT32 nMajorVersion, OUT smiLPUINT32 nMinorVersion, OUT smiLPUINT32 nLevel, OUT smiLPUINT32 nTranslateMode, OUT smiLPUINT32 nRetransmitMode);
SNMPAPI_STATUS	SnmpCleanup	(void);
HSNMP_SESSION	SnmpCreateSession	(IN HWND hWnd, IN UINT wMsg, IN CALLBACK fCallBack, IN LPVOID lpClientData);
smiINT	SnmpSelect	(IN smiINT nfds, IN fd_set *readfds, IN fd_set *writefds, IN fd_set *exceptfds, IN struct timeval *timeout);
SNMPAPI_STATUS	SnmpClose	(IN HSNMP_SESSION session);

Return Type	Procedure Name	Parameters
SNMPAPI_STATUS	SnmpSendMsg	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN HSNMP_PDU pdu);
SNMPAPI_STATUS	SnmpRecvMsg	(IN HSNMP_SESSION session, OUT LPHSNMP_ENTITY srcEntity, OUT LPHSNMP_ENTITY dstEntity, OUT LPHSNMP_CONTEXT context OUT LPHSNMP_PDU pdu);
SNMPAPI_STATUS	SnmpRegister	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN smiLPCOID notification, IN smiUINT32 state);

Entity/Context Functions

The functions in this section enable the application to use human-oriented string identifiers for the entity and context objects and concepts, while permitting the WinSNMP implementation to adopt proprietary repository, access method, and runtime representation strategies.

The functions in this section are:

Return Type	Procedure Name	Parameters
HSNMP_ENTITY	SnmpStrToEntity	(IN HSNMP_SESSION session, IN LPCSTR entity);
SNMPAPI_STATUS	SnmpEntityToStr	(IN HSNMP_ENTITY entity, IN smiUINT32 size, OUT LPSTR string);
SNMPAPI_STATUS	SnmpFreeEntity	(IN HSNMP_ENTITY entity);
HSNMP_CONTEXT	SnmpStrToContext	(IN HSNMP_SESSION session, IN smiLPCOCTETS string);
SNMPAPI_STATUS	SnmpContextToStr	(IN HSNMP_CONTEXT context, OUT smiLPOCTETS string);
SNMPAPI_STATUS	SnmpFreeContext	(IN HSNMP_CONTEXT context);

PDU Functions

This section defines functions which construct PDUs for use in the SnmpSendMsg and SnmpEncodeMsg functions and which decompose PDUs received via the SnmpRecvMsg and SnmpDecodeMsg functions. The variable binding functions also pertain to PDU composition and decomposition, but are retained as a separate section.

Actual PDU and variable binding data structures are private to the WinSNMP implementation. The PDU and variable binding functions enable applications to extract the component data elements that are available for whatever use the application deems appropriate. The elements comprising a PDU from the perspective of a WinSNMP application are:

```
/* This typedef is for expository purposes only.
It is not a required component of WinSNMP */
typedef struct {
        smiINT       PDU_type;
        smiINT32     request_id;
        smiINT       error_status   -- "non_repeaters" for BulkPDU
        smiINT       error_index    -- "max_repetitions" for BulkPDU
        HSNMP_VBL    varbindlist;}  -- we'll examine this one in the next section
                     PDU;
```

The functions in this section are:

Return Type	**Procedure Name**	**Parameters**
HSNMP_PDU	SnmpCreatePdu	(IN HSNMP_SESSION session, IN smiINT PDU_type, IN smiINT32 request_id, IN smiINT error_status/non_repeaters, IN smiINT error_index/max_repetitions, IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpGetPduData	(IN HSNMP_PDU PDU, OUT smiLPINT PDU_type, OUT smiLPINT32 request_id, OUT smiLPINT error_status/non_repeaters, OUT smiLPINT error_index/max_repetitions, OUT LPHSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpSetPduData	(IN HSNMP_PDU PDU, IN const smiINT FAR *PDU_type, IN const smiINT32 FAR *request_id, IN const smiINT FAR *non_repeaters, IN const smiINT FAR *max_repetitions, IN const HSNMP_VBL FAR *vbl);
HSNMP_PDU	SnmpDuplicatePdu	(IN HSNMP_SESSION session, IN HSNMP_PDU PDU);
SNMPAPI_STATUS	SnmpFreePdu	(IN HSNMP_PDU PDU);

The following table illustrates the possible PDU_type values used in WinSNMP functions:

PDU_types Table

SNMP_PDU_GET	Indicates a Get Request-PDU
SNMP_PDU_GETNEXT	Indicates a GetNextRequest-PDU
SNMP_PDU_GETBULK	Indicates a GetBulkRequest-PDU
SNMP_PDU_V1TRAP	Indicates an SNMPv1-Trap-PDU
SNMP_PDU_SET	Indicates a SetRequest- PDU
SNMP_PDU_INFORM	Indicates an InformRequest-PDU
SNMP_PDU_RESPONSE	Indicates a Response-PDU
SNMP_PDU_TRAP	Indicates an SNMPv2-Trap-PDU

The following table illustrates the possible SNMP error values used in the error_status element of an SNMP PDU:

SNMP Error Values Table

SNMP_ERROR_NOERROR	Specifies the noError error.
SNMP_ERROR_TOOBIG	Specifies the tooBig error.
SNMP_ERROR_NOSUCHNAME	Specifies the noSuchName error.
SNMP_ERROR_BADVALUE	Specifies the badValue error.
SNMP_ERROR_READONLY	Specifies the readOnly error.
SNMP_ERROR_GENERR	Specifies the genErr error.
SNMP_ERROR_NOACCESS	Specifies the noAccess error.
SNMP_ERROR_WRONGTYPE	Specifies the wrongType error.
SNMP_ERROR_WRONGLENGTH	Specifies the wrongLength error.
SNMP_ERROR_WRONGENCODING	Specifies the wrongEncoding error.
SNMP_ERROR_WRONGVALUE	Specifies the wrongValue error.
SNMP_ERROR_NOCREATION	Specifies the noCreation error.
SNMP_ERROR_INCONSISTENTVALUE	Specifies the inconsistentValue error.
SNMP_ERROR_RESOURCEUNAVAILABLE	Specifies the resourceUnavailable error.
SNMP_ERROR_COMMITFAILED	Specifies the commitFailed error.
SNMP_ERROR_UNDOFAILED	Specifies the undoFailed error.
SNMP_ERROR_AUTHORIZATIONERROR	Specifies the authorizationError error.
SNMP_ERROR_NOTWRITABLE	Specifies the notWritable error.
SNMP_ERROR_INCONSISTENTNAME	Specifies the inconsistentName error.

Variable Binding Functions

WinSNMP relies on a varbindlist structure (VBL), and drops the concept of a separate varbind structure (VB). No capability is lost because an individual varbind structure can be represented by a varbindlist structure of one member. A WinSNMP *application* accesses the varbindlist structure via handles of type HSNMP_VBL. A WinSNMP implementation hides the details of this structure from the application using whatever proprietary mechanisms and techniques it considers optimal.

These functions allow applications to easily construct and manipulate VarBindLists for inclusion in PDUs. Note that a varbind is not directly associated with a PDU, only indirectly through inclusion in a varbindlist. A varbindlist gets associated with and de-referenced from a PDU with the SnmpSetPduData and SnmpGetPduData, respectively.

The functions in this section are:

Return Type	Procedure Name	Parameters
HSNMP_VBL	SnmpCreateVbl	(IN HSNMP_SESSION session, IN smiLPCOID name, IN smiLPCVALUE value);

Return Type	Procedure Name	Parameters
HSNMP_VBL	SnmpDuplicateVbl	(IN HSNMP_SESSION session, IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpFreeVbl	(IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpCountVbl	(IN HSNMP_VBL vbl);
SNMPAPI_STATUS	SnmpGetVb	(IN HSNMP_VBL vbl, IN smiUINT32 index, OUT smiLPOID name, OUT smiLPVALUE value);
SNMPAPI_STATUS	SnmpSetVb	(IN HSNMP_VBL vbl, IN smiUINT32 index, IN smiLPCOID name, IN smiLPCVALUE value);
SNMPAPI_STATUS	SnmpDeleteVb	(IN HSNMP_VBL vbl, IN smiUINT32 index);

Table of Syntax Values Used in Variable Binding Data Structures

SNMP_SYNTAX_INT32
SNMP_SYNTAX_OCTETS
SNMP_SYNTAX_OID
SNMP_SYNTAX_BITS
SNMP_SYNTAX_IPADDR
SNMP_SYNTAX_CNTR32
SNMP_SYNTAX_GAUGE32
SNMP_SYNTAX_TIMETICKS
SNMP_SYNTAX_OPAQUE
SNMP_SYNTAX_NSAPADDR
SNMP_SYNTAX_CNTR64
SNMP_SYNTAX_UINT32
SNMP_SYNTAX_NULL
SNMP_SYNTAX_NOSUCHOBJECT
SNMP_SYNTAX_NOSUCHINSTANCE
SNMP_SYNTAX_ENDOFMIBVIEW

Utility Functions

The utility functions are offered to ease the tasks of bookkeeping and dealing with objects passed across the Windows SNMP interface.

Return Type	Procedure Name	Parameters
SNMPAPI_STATUS	SnmpGetLastError	(IN HSNMP_SESSION session);
smiLPBYTE	SnmpGetLastErrorStr	(IN HSNMP_SESSION session);
SNMPAPI_STATUS	SnmpStrToOid	(IN LPCSTR string, OUT smiLPOID dstOID);
SNMPAPI_STATUS	SnmpOidToStr	(IN smiLPCOID srcOID, IN smiUINT32 size, OUT LPSTR string);
SNMPAPI_STATUS	SnmpOidCopy	(IN smiLPCOID srcOID, OUT smiLPOID dstOID);

Return Type	**Procedure Name**	**Parameters**
SNMPAPI_STATUS	SnmpOidCompare	(IN smiLPCOID xOID, IN smiLPCOID yOID, IN smiUINT32 maxlen, OUT smiLPINT result);
SNMPAPI_STATUS	SnmpEncodeMsg	(IN HSNMP_SESSION session, IN HSNMP_ENTITY srcEntity, IN HSNMP_ENTITY dstEntity, IN HSNMP_CONTEXT context, IN HSNMP_PDU pdu, OUT smiLPOCTETS msgBufDesc);
SNMPAPI_STATUS	SnmpDecodeMsg	(IN HSNMP_SESSION session, OUT LPHSNMP_ENTITY srcEntity, OUT LPHSNMP_ENTITY dstEntity, OUT LPHSNMP_CONTEXT context, OUT LPHSNMP_PDU pdu, IN smiLPCOCTETS msgBufDesc);
SNMPAPI_STATUS	SnmpFreeDescriptor	(IN smiUINT32 syntax, IN smiLPOPAQUE descriptor);

Declarations

The WinSNMP.h *include* file contains common declarations for SNMP datatypes, attributes, and values and for WinSNMP API datatypes, attributes, and values. It *must* be delivered as WinSNMP.h with every compliant implementation.

Additional declarations required or offered by an implementation must be delivered in a separate include file with an implementation-specific name.

An attempt has been made to balance brevity and clarity in these declarations. In general, however, there has been a slight bias toward brevity. Developers can easily include longer, more descriptive equivalents to the declarations through additional *#define* and *typedef* statements in a private include files loaded after WinSNMP.h.

WinSNMP.h Include File

```
/* v1.1 WinSNMP.h */
/* v1.0 - Sep 13, 1993 */
/* v1.1 - Jun 10, 1994 */

#ifndef _INC_WINSNMP        /*Include WinSNMP declarations */
#define  _INC_WINSNMP       /* Just once! */

#ifndef _INC_WINDOWS        /* Include Windows declarations, if not already done */
#include <windows.h>
#define _INC_WINDOWS        /* Just once! */
#endif                      /* _INC_WINDOWS */

#ifdef __cplusplus
extern "C" {
#endif

/* WinSNMP API Type Definitions  */
typedef  HANDLE             HSNMP_SESSION,   FAR *LPHSNMP_SESSION;
typedef  HANDLE             HSNMP_ENTITY,    FAR *LPHSNMP_ENTITY;
typedef  HANDLE             HSNMP_CONTEXT,   FAR *LPHSNMP_CONTEXT;
typedef  HANDLE             HSNMP_PDU,       FAR *LPHSNMP_PDU;
typedef  HANDLE             HSNMP_VBL,       FAR *LPHSNMP_VBL;
typedef unsigned char       smiBYTE,         FAR *smiLPBYTE;
/* SNMP-related types from RFC1442 (SMI) */
typedef signed long         smiINT,          FAR *smiLPINT;
typedef smiINT              smiINT32,        FAR *smiLPINT32;
typedef unsigned long       smiUINT32,       FAR *smiLPUINT32;
typedef struct {
   smiUINT32 len;
   smiLPBYTE ptr;}          smiOCTETS,       FAR *smiLPOCTETS;
typedef const smiOCTETS                      FAR *smiLPCOCTETS;
typedef smiOCTETS           smiBITS,         FAR *smiLPBITS;
typedef struct {
   smiUINT32   len;
   smiLPUINT32 ptr;}        smiOID,          FAR *smiLPOID;
typedef const smiOID                         FAR *smiLPCOID;
typedef smiOCTETS           smiIPADDR,       FAR *smiLPIPADDR;
typedef smiUINT32           smiCNTR32,       FAR *smiLPCNTR32;
typedef smiUINT32           smiGAUGE32,      FAR *smiLPGAUGE32;
typedef smiUINT32           smiTIMETICKS,    FAR *smiLPTIMETICKS;
typedef smiOCTETS           smiOPAQUE,       FAR *smiLPOPAQUE;
typedef smiOCTETS           smiNSAPADDR,     FAR *smiLPNSAPADDR;
typedef struct {
        smiUINT32 hipart;
        smiUINT32 lopart;}  smiCNTR64,       FAR *smiLPCNTR64;

/* ASN/BER Base Types */
/* (used in forming SYNTAXes and certain SNMP types/values) */
#define ASN_UNIVERSAL                (0x00)
#define ASN_APPLICATION              (0x40)
#define ASN_CONTEXT                  (0x80)
#define ASN_PRIVATE                  (0xC0)
#define ASN_PRIMITIVE                (0x00)
#define ASN_CONSTRUCTOR              (0x20)
```

```
/* SNMP ObjectSyntax Values */
#define SNMP_SYNTAX_SEQUENCE  (ASN_UNIVERSAL | ASN_CONSTRUCTOR | 0x10)
/* These values are used in the "syntax" member of the smiVALUE structure which follows */
#define SNMP_SYNTAX_INT       (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x02)
#define SNMP_SYNTAX_BITS      (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x03)
#define SNMP_SYNTAX_OCTETS    (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x04)
#define SNMP_SYNTAX_NULL      (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x05)
#define SNMP_SYNTAX_OID       (ASN_UNIVERSAL | ASN_PRIMITIVE | 0x06)
#define SNMP_SYNTAX_INT32     SNMP_SYNTAX_INT
#define SNMP_SYNTAX_IPADDR    (ASN_APPLICATION | ASN_PRIMITIVE | 0x00)
#define SNMP_SYNTAX_CNTR32    (ASN_APPLICATION | ASN_PRIMITIVE | 0x01)
#define SNMP_SYNTAX_GAUGE32   (ASN_APPLICATION | ASN_PRIMITIVE | 0x02)
#define SNMP_SYNTAX_TIMETICKS (ASN_APPLICATION | ASN_PRIMITIVE | 0x03)
#define SNMP_SYNTAX_OPAQUE    (ASN_APPLICATION | ASN_PRIMITIVE | 0x04)
#define SNMP_SYNTAX_NSAPADDR  (ASN_APPLICATION | ASN_PRIMITIVE | 0x05)
#define SNMP_SYNTAX_CNTR64    (ASN_APPLICATION | ASN_PRIMITIVE | 0x06)
#define SNMP_SYNTAX_UINT32    (ASN_APPLICATION | ASN_PRIMITIVE | 0x07)
/* Exception conditions in response PDUs for SNMPv2 */
#define SNMP_SYNTAX_NOSUCHOBJECT    (ASN_CONTEXT | ASN_PRIMITIVE | 0x00)
#define SNMP_SYNTAX_NOSUCHINSTANCE  (ASN_CONTEXT | ASN_PRIMITIVE | 0x01)
#define SNMP_SYNTAX_ENDOFMIBVIEW    (ASN_CONTEXT | ASN_PRIMITIVE | 0x02)

typedef struct {              /* smiVALUE portion of VarBind */
        smiUINT32   syntax;   /* Insert SNMP_SYNTAX_<type> */
        union {
        smiINT      sNumber;  /* SNMP_SYNTAX_INT
                                 SNMP_SYNTAX_INT32 */
        smiUINT32   uNumber;  /* SNMP_SYNTAX_UINT32
                                 SNMP_SYNTAX_CNTR32
                                 SNMP_SYNTAX_GAUGE32
                                 SNMP_SYNTAX_TIMETICKS */
        smiCNTR64   hNumber;  /* SNMP_SYNTAX_CNTR64 */
        smiOCTETS   string;   /* SNMP_SYNTAX_OCTETS
                                 SNMP_SYNTAX_BITS
                                 SNMP_SYNTAX_OPAQUE
                                 SNMP_SYNTAX_IPADDR
                                 SNMP_SYNTAX_NSAPADDR */
        smiOID      oid;      /* SNMP_SYNTAX_OID */
        smiBYTE     empty;    /* SNMP_SYNTAX_NULL
                                 SNMP_SYNTAX_NOSUCHOBJECT
                                 SNMP_SYNTAX_NOSUCHINSTANCE
                                 SNMP_SYNTAX_ENDOFMIBVIEW */
            }       value;    /* union */
        }           smiVALUE, FAR *smiLPVALUE;
typedef const smiVALUE        FAR *smiLPCVALUE;

/* SNMP Limits   */
#define MAXOBJIDSIZE      128   /* Max number of components in an OID */
#define MAXOBJIDSTRSIZE   1408  /* Max len of decoded MAXOBJIDSIZE OID */
```

```
/* PDU Type Values */
#define SNMP_PDU_GET            (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x0)
#define SNMP_PDU_GETNEXT        (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x1)
#define SNMP_PDU_RESPONSE       (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x2)
#define SNMP_PDU_SET            (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x3)
/* SNMP_PDU_V1TRAP is obsolete in SNMPv2 */
#define SNMP_PDU_V1TRAP         (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x4)
#define SNMP_PDU_GETBULK        (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x5)
#define SNMP_PDU_INFORM         (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x6)
#define SNMP_PDU_TRAP           (ASN_CONTEXT | ASN_CONSTRUCTOR | 0x7

/* SNMPv1 Trap Values */
/* (These values might be superfluous wrt WinSNMP applications) */
#define SNMP_TRAP_COLDSTART             0
#define SNMP_TRAP_WARMSTART             1
#define SNMP_TRAP_LINKDOWN              2
#define SNMP_TRAP_LINKUP                3
#define SNMP_TRAP_AUTHFAIL              4
#define SNMP_TRAP_EGPNEIGHBORLOSS       5
#define SNMP_TRAP_ENTERPRISESPECIFIC    6

/* SNMP Error Codes Returned in Error_status Field of PDU */
/* (these are NOT WinSNMP API Error Codes */
/* Error Codes Common to SNMPv1 and SNMPv2 */
#define SNMP_ERROR_NOERROR              0
#define SNMP_ERROR_TOOBIG               1
#define SNMP_ERROR_NOSUCHNAME           2
#define SNMP_ERROR_BADVALUE             3
#define SNMP_ERROR_READONLY             4
#define SNMP_ERROR_GENERR               5
/* Error Codes Added for SNMPv2 */
#define SNMP_ERROR_NOACCESS             6
#define SNMP_ERROR_WRONGTYPE            7
#define SNMP_ERROR_WRONGLENGTH          8
#define SNMP_ERROR_WRONGENCODING        9
#define SNMP_ERROR_WRONGVALUE           10
#define SNMP_ERROR_NOCREATION           11
#define SNMP_ERROR_INCONSISTENTVALUE    12
#define SNMP_ERROR_RESOURCEUNAVAILABLE  13
#define SNMP_ERROR_COMMITFAILED         14
#define SNMP_ERROR_UNDOFAILED           15
#define SNMP_ERROR_AUTHORIZATIONERROR   16
#define SNMP_ERROR_NOTWRITABLE          17
#define SNMP_ERROR_INCONSISTENTNAME     18
```

```
/* WinSNMP API Values */
/* Values used to indicate entity/context translation modes */
#define SNMPAPI_TRANSLATED          0
#define SNMPAPI_UNTRANSLATED_V1     1
#define SNMPAPI_UNTRANSLATED_V2     2

/* Values used to indicate SNMP "communications level" supported by the implementation */
#define SNMPAPI_NO_SUPPORT          0
#define SNMPAPI_V1_SUPPORT          1
#define SNMPAPI_V2_SUPPORT          2
#define SNMPAPI_M2M_SUPPORT         3

/* Values used to indicate retransmit mode in the implementation */
#define SNMPAPI_OFF                 0     /* Refuse support */
#define SNMPAPI_ON                  1     /* Request support */

/* WinSNMP API Function Return Codes */
typedef smiUINT32        SNMPAPI_STATUS   /* Used for function ret values */
#define SNMPAPI_FAILURE             0     /* Generic error code */
#define SNMPAPI_SUCCESS             1     /* Generic success code */
```

```
/* WinSNMP API Error Codes (for SnmpGetLastError) */
/* (NOT SNMP Response-PDU error_status codes) */
#define SNMPAPI_ALLOC_ERROR         2    /* Error allocating memory */
#define SNMPAPI_CONTEXT_INVALID     3    /* Invalid context parameter */
#define SNMPAPI_CONTEXT_UNKNOWN     4    /* Unknown context parameter */
#define SNMPAPI_ENTITY_INVALID      5    /* Invalid entity parameter */
#define SNMPAPI_ENTITY_UNKNOWN      6    /* Unknown entity parameter */
#define SNMPAPI_INDEX_INVALID       7    /* Invalid VBL index parameter */
#define SNMPAPI_NOOP                8    /* No operation performed */
#define SNMPAPI_OID_INVALID         9    /* Invalid OID parameter */
#define SNMPAPI_OPERATION_INVALID   10   /* Invalid/unsupported operation */
#define SNMPAPI_OUTPUT_TRUNCATED    11   /* Insufficient output buf len */
#define SNMPAPI_PDU_INVALID         12   /* Invalid PDU parameter */
#define SNMPAPI_SESSION_INVALID     13   /* Invalid session parameter */
#define SNMPAPI_SYNTAX_INVALID      14   /* Invalid syntax in smiVALUE */
#define SNMPAPI_VBL_INVALID         15   /* Invalid VBL parameter */
#define SNMPAPI_MODE_INVALID        16   /* Invalid mode parameter */
#define SNMPAPI_SIZE_INVALID        17   /* Invalid size/length parameter */
#define SNMPAPI_NOT_INITIALIZED     18   /* SnmpStartup failed/not called */
#define SNMPAPI_MESSAGE_INVALID     19   /* Invalid SNMP message format */
#define SNMPAPI_HWND_INVALID        20   /* Invalid Window handle */
#define SNMPAPI_OTHER_ERROR         99   /* For internal/undefined errors */
/* Generic Transport Layer (TL) Errors */
#define SNMPAPI_TL_NOT_INITIALIZED  100  /* TL not initialized */
#define SNMPAPI_TL_NOT_SUPPORTED    101  /* TL does not support protocol */
#define SNMPAPI_TL_NOT_AVAILABLE    102  /* Network subsystem has failed */
#define SNMPAPI_TL_RESOURCE_ERROR   103  /* TL resource error */
#define SNMPAPI_TL_UNDELIVERABLE    104  /* Destination unreachable */
#define SNMPAPI_TL_SRC_INVALID      105  /* Source endpoint invalid */
#define SNMPAPI_TL_INVALID_PARAM    106  /* Input parameter invalid */
#define SNMPAPI_TL_IN_USE           107  /* Source endpoint in use */
#define SNMPAPI_TL_TIMEOUT          108  /* No response before timeout */
#define SNMPAPI_TL_PDU_TOO_BIG      109  /* PDU too big for send/receive */
#define SNMPAPI_TL_OTHER            199  /* Undefined TL error */

/* WinSNMP API Function Prototypes */
#define IN                               /* Documentation only */
#define OUT                              /* Documentation only */
#define SNMPAPI_CALL    WINAPI           /* FAR PASCAL calling conventions */
```

```
/* Local Database Functions */

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetTranslateMode
                                  (OUT smiLPUINT32 nTranslateMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetTranslateMode
                                  (IN smiUINT32 nTranslateMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetRetransmitMode
                                  (OUT smiLPUINT32 nRetransmitMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetRetransmitMode
                                  (IN smiUINT32 nRetransmitMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetTimeout
                                  (IN HSNMP_ENTITY hEntity,
                                  OUT smiLPTIMETICKS nPolicyTimeout,
                                  OUT smiLPTIMETICKS nActualTimeout);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetTimeout
                                  (IN HSNMP_ENTITY hEntity,
                                  IN smiTIMETICKS nPolicyTimeout);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetRetry
                                  (IN HSNMP_ENTITY hEntity,
                                  OUT smiLPUINT32 nPolicyRetry,
                                  OUT smiLPUINT32 nActualRetry);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetRetry
                                  (IN HSNMP_ENTITY hEntity,
                                  IN smiUINT32 nPolicyRetry);
```

```
/* Communications Functions */

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpStartup
                                  (OUT smiLPUINT32 nMajorVersion,
                                  OUT smiLPUINT32 nMinorVersion,
                                  OUT smiLPUINT32 nLevel,
                                  OUT smiLPUINT32 nTranslateMode,
                                  OUT smiLPUINT32 nRetransmitMode);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpCleanup
                                  (void);

HSNMP_SESSION     SNMPAPI_CALL    SnmpCreateSession
                                  (IN HWND hWnd,
                                  IN UINT wMsg,
                                  IN CALLBACK fCallBack,
                                  IN LPVOID lpclientData);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpClose
                                  (IN HSNMP_SESSION session);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSendMsg
                                  (IN HSNMP_SESSION session,
                                  IN HSNMP_ENTITY srcEntity,
                                  IN HSNMP_ENTITY dstEntity,
                                  IN HSNMP_CONTEXT context,
                                  IN HSNMP_PDU PDU);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpRecvMsg
                                  (IN HSNMP_SESSION session,
                                  OUT LPHSNMP_ENTITY srcEntity,
                                  OUT LPHSNMP_ENTITY dstEntity,
                                  OUT LPHSNMP_CONTEXT  context,
                                  OUT LPHSNMP_PDU PDU);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpRegister
                                  (IN HSNMP_SESSION session,
                                  IN HSNMP_ENTITY srcEntity,
                                  IN HSNMP_ENTITY dstEntity,
                                  IN HSNMP_CONTEXT context,
                                  IN smiLPCOID notification,
                                  IN smiUINT32 state);
```

```
/* Entity/Context Functions */

HSNMP_ENTITY      SNMPAPI_CALL    SnmpStrToEntity
                                  (IN HSNMP_SESSION session,
                                  IN LPCSTR string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpEntityToStr
                                  (IN HSNMP_ENTITY entity,
                                  IN smiUINT32 size,
                                  OUT LPSTR string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpFreeEntity
                                  (IN HSNMP_ENTITY entity);

HSNMP_CONTEXT     SNMPAPI_CALL    SnmpStrToContext
                                  (IN HSNMP_SESSION session,
                                  IN smiLPCOCTETS string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpContextToStr
                                  (IN HSNMP_CONTEXT context,
                                  OUT smiLPOCTETS string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpFreeContext
                                  (IN HSNMP_CONTEXT context);
/* PDU Functions */

HSNMP_PDU         SNMPAPI_CALL    SnmpCreatePdu
                                  (IN HSNMP_SESSION session,
                                  IN smiINT PDU_type,
                                  IN smiINT32 request_id,
                                  IN smiINT error_status,
                                  IN smiINT error_index,
                                  IN HSNMP_VBL varbindlist);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetPduData
                                  (IN HSNMP_PDU PDU,
                                  OUT smiLPINT PDU_type,
                                  OUT smiLPINT32 request_id,
                                  OUT smiLPINT error_status,
                                  OUT smiLPINT error_index,
                                  OUT LPHSNMP_VBL varbindlist);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpSetPduData
                                  (IN HSNMP_PDU PDU,
                                  IN const smiINT FAR *PDU_type,
                                  IN const smiINT32 FAR *request_id,
                                  IN const smiINT FAR *non_repeaters,
                                  IN const smiINT FAR *max_repetitions,
                                  IN const HSNMP_VBL FAR *varbindlist);

HSNMP_PDU         SNMPAPI_CALL    SnmpDuplicatePdu
                                  (IN HSNMP_SESSION session,
                                  IN HSNMP_PDU PDU);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpFreePdu
                                  (IN HSNMP_PDU PDU);
```

```
/* Variable-Binding Functions */

HSNMP_VBL          SNMPAPI_CALL    SnmpCreateVbl
                                   (IN HSNMP_SESSION session,
                                   IN smiLPCOID name,
                                   IN smiLPCVALUE value);

HSNMP_VBL          SNMPAPI_CALL    SnmpDuplicateVbl
                                   (IN HSNMP_SESSION session,
                                   IN HSNMP_VBL vbl);

SNMPAPI_STATUS     SNMPAPI_CALL    SnmpFreeVbl
                                   (IN HSNMP_VBL vbl);

SNMPAPI_STATUS     SNMPAPI_CALL    SnmpCountVbl
                                   (IN HSNMP_VBL vbl);

SNMPAPI_STATUS     SNMPAPI_CALL    SnmpGetVb
                                   (IN HSNMP_VBL vbl,
                                   IN smiUINT32 index,
                                   OUT smiLPOID name,
                                   OUT smiLPVALUE value);

SNMPAPI_STATUS     SNMPAPI_CALL    SnmpSetVb
                                   (IN HSNMP_VBL vbl,
                                   IN smiUINT32 index,
                                   IN smiLPCOID name,
                                   IN smiLPCVALUE value);

SNMPAPI_STATUS     SNMPAPI_CALL    SnmpDeleteVb
                                   (IN HSNMP_VBL vbl,
                                   IN smiUINT32 index);
```

```
/* Utility Functions */

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpGetLastError
                                  (IN HSNMP_SESSION session);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpStrToOid
                                  (IN LPCSTR string,
                                  OUT smiLPOID dstOID);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidToStr
                                  (IN smiLPCOID srcOID,
                                  IN smiUINT32 size,
                                  OUT LPSTR string);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidCopy
                                  (IN smiLPCOID srcOID,
                                  OUT smiLPOID dstOID);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpOidCompare
                                  (IN smiLPCOID xOID,
                                  IN smiLPCOID yOID,
                                  IN smiUINT32 maxlen,
                                  OUT smiLPINT result);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpEncodeMsg
                                  (IN HSNMP_SESSION session,
                                  IN HSNMP_ENTITY srcEntity,
                                  IN HSNMP_ENTITY dstEntity,
                                  IN HSNMP_CONTEXT context,
                                  IN HSNMP_PDU pdu,
                                  OUT smiLPOCTETS msgBufDesc);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpDecodeMsg
                                  (IN HSNMP_SESSION session,
                                  OUT LPHSNMP_ENTITY srcEntity,
                                  OUT LPHSNMP_ENTITY dstEntity,
                                  OUT LPHSNMP_CONTEXT context,
                                  OUT LPHSNMP_PDU pdu,
                                  IN smiLPCOCTETS msgBufDesc);

SNMPAPI_STATUS    SNMPAPI_CALL    SnmpFreeDescriptor
                                  (IN smiUINT32 syntax,
                                  IN smiLPOPAQUE descriptor);

#ifdef __cplusplus
}
#endif
#endif     /* _INC_WINSNMP */
```

Mapping Traps Between SNMPv1 and SNMPv2

One of the differences between SNMPv1 and SNMPv2 is a change to the trap format: in SNMPv1, the trap format was unlike the format of the other PDUs; in SNMPv2 the trap format is identical to the format of the other PDUs.

When the WinSNMP API delivers a trap to a management application, it always uses the SNMPv2 trap format, even if an SNMPv1 agent generated the trap. The SNMPv2 coexistence document, RFC 1908, specifies how an SNMPv1 trap is translated into the SNMPv2 trap format, and this algorithm is used in all implementations of the WinSNMP API.

In SNMPv1, the trap format has five fields:

Enterprise	Identifies the type of device that generated the trap
Agent-addr	Identifies the network address of the device
Generic-trap/specific-trap	Identifies the trap that was generated
Time-stamp	Identifies when the trap was generated
Variable-bindings	Contain the payload, if any, associated with the trap.

In SNMPv2, the trap format consists simply of a list of “n” variable bindings:

- The first variable binding contains the time-stamp
- The second variable binding identifies the trap, using an OID
- The third through “n” variable bindings, if any, contain the payload.

When SnmpRecvMsg returns an SNMP_PDU_TRAP message, the application examines the variable bindings of that message to determine trap information.

When translating an SNMPv1 trap to the SNMPv2 format, one additional variable binding may be present, at the end of the list, which corresponds to the enterprise field. According to the SNMPv2 coexistence document, this variable binding need only be present if the trap was enterprise-specific. However, in order to simplify the programming of management applications, this variable binding is always added by the WinSNMP API when it translates an SNMPv1 trap to the SNMPv2 format.

Chapter 17. Using the NetView for AIX SNMP API

This chapter describes the NetView for AIX Simple Network Management Protocol Application Programming Interface (SNMP API). It includes the following:

- An overview of SNMP
- The functions and key data structures in the SNMP API
- How to perform basic tasks using the SNMP API

The detailed reference pages and error code information for the SNMP API are in the *NetView for AIX Programmer's Reference* and the online man pages.

This chapter is for application developers who need to use the SNMP API. Readers are presumed to have the following background:

- Programming skills using the C programming language. In addition to general programming skills, the SNMP API programmer should understand the principles of memory management (allocation and deallocation) and the use of select() or a similar I/O mechanism.
- General networking familiarity. In particular, readers should understand these concepts:
  - Transport reliability, and other developer-visible differences between UDP and TCP
  - Internet addressing
- Substantial knowledge of network management principles, the Simple Network Management Protocol, and related topics. In particular, readers should be familiar with the following:
  - The SNMP protocol definition as described in RFC 1157
  - The Internet-standard Management Information Base (MIB) as described in RFC 1155 and RFC 1213
  - The concepts of managers and agents

The SNMP Model of Communication

Managing contemporary computer networks requires an approach that simplifies the potentially complex problems of communication and coordination. The prevailing approach, and the one adopted by SNMP, is to view the network as a collection of cooperative, communicating entities. There are basically two types of entities: management nodes (*managers*) and managed nodes (*agents*).

Managers

A manager is a node that actively participates in network management. It solicits and interprets data about network devices and network traffic, and typically interacts with a user to achieve the user's intentions. A manager can also trigger changes in an agent by changing the value of a variable on the agent node. Managers are frequently implemented as network management applications.

Agents

An SNMP agent is software that resides on a network node and is responsible for communicating with managers regarding that node. The node is represented as a managed object, which has various fields or variables, which are defined in the appropriate MIB. The agent has two purposes:

- To respond to requests from managers, supplying or changing the values of the object's variables as requested
- To generate traps to alert managers of noteworthy events occurring at the node, such as a component failure

Not all devices support SNMP directly. A device that does not directly support SNMP is called *foreign*. A *proxy agent* is an agent that serves a foreign device by translating between SNMP and the foreign device's protocol.

Manager and Agent Interaction

A manager can be one of many processes on a specific computer. For example, a manager might try to provide data about certain gateways. It would use the gateway agent by requesting data about throughput, retransmission rates, and other parameters. Such a manager might also need to reset gateway counters periodically by communicating with the gateway agent.

Managers do not need to know any internal details about the object managed by an agent. Likewise, an SNMP agent can service requests from many SNMP managers. The agent does not need to know the context of the request or the structure of the manager making the request. The agent validates the request, services it, and enters its passive state awaiting the next request. This division of responsibilities simplifies network management solutions.

Note: The NetView for AIX SNMP API is intended for the development of SNMP-based network management applications. It does not support the development of SNMP agents, other than allowing the generation of traps.

SNMP Messages

Requests and responses are transferred in Protocol Data Units, or PDUs. A PDU is the formal name for a message that is sent or received in the course of SNMP communication.

SNMP uses the User Datagram Protocol (UDP), an intrinsically connectionless channel. This implementation of an SNMP API is based on Revision 1.1 of the CMU SNMP Library, which introduces a connection-oriented model from the application perspective. Internally, this implementation still uses connectionless UDP; the agent is not involved in opening or closing a session.

Types of Messages

Table 27 describes the five basic types of messages your application can handle.

Table 27. SNMP Message Types

Type	Formal Name	Description
Get Request	`GET_REQ_MSG`	A Get Request message requests the value of one or more of the variables of the object managed by an agent.
Set Request	`SET_REQ_MSG`	A Set Request message writes new data to one or more of the variables of the object managed by an agent.
Get Next Request	`GETNEXT_REQ_MSG`	A Get Next Request message requests the Object Identifier(s) and value(s) of the next variable of the object managed by an agent.
Trap Request	`TRAP_REQ_MSG`	A Trap Request message sends a non-blocking alert to an SNMP manager. Traps have special semantics and values.
Get Response	`GET_RSP_MSG`	A Get Response message contains data that has come from an agent in response to a Get Request.

Traps

In the model presented previously, the manager requests information from the agent. The agent then responds. However, it is possible for an agent to issue messages without a corresponding request. Such a message is known as a *trap*.

Traps exist to handle special conditions. When an agent or manager detects such a condition, it can emit a trap message. There are several predefined traps specified by SNMP (see the /usr/OV/include/OV/OVsnmpApi.h file), which can be extended by using enterprise-specific traps. Using the NetView for AIX SNMP API, an application can receive SNMP traps.

The Management Information Base

This section reviews highlights of data representation and the concept of a Management Information Base, or MIB.

The MIB is a method of describing managed objects by specifying the names, types, and order of the fields, or variables, that make up the object. A MIB contains the definitions for a collection of standardized and non-standardized (vendor, experimental) objects.

The Internet MIB-II is one of many standard MIBs. The purpose of the MIB-II is to define common objects for managing TCP/IP networks. Other standard MIBs exist (or are being defined) to manage specific network elements as well.

Note: Detailed explanations of these topics are beyond the scope of this guide. For more information about these topics, refer to RFC 1155, RFC 1212, and RFC 1213. Also see "For Further Reading" on page 288 in this chapter.

The Internet MIB-II definition (RFC 1213) defines standardized objects for TCP/IP agents. To access the value of a MIB-II object, an SNMP manager sends a request to the agent representing the desired instance of the object. The request message contains MIB information (an object identifier) that lets the agent identify the specific objects. The corresponding response message from the agent carries the same identifying information.

Object Identifiers

For the purpose of developing SNMP applications, an *object identifier* (OID) is a data type that precisely identifies a MIB-II object. An OID (sometimes referred to as the *registration ID*) consists of a sequence of non-negative integers that describe a path through the object-naming hierarchy to the object. The naming hierarchy is commonly called the *naming tree*.

The Naming Tree

The naming tree has the structure of a conventional tree with arbitrary breadth and depth. The nodes are labeled with non-negative integers (each node among siblings must have a unique label).

Various organizations have administrative authority for assigning labels within subtrees of the naming tree. They can assign subordinate, or *child*, nodes, and/or delegate this responsibility to still other organizations. The root node of the naming tree has three children:

`ccitt(0)`	The administration authority for this branch is the International Telegraph and Telephone Consultative Committee (CCITT).
`iso(1)`	The administration authority for this branch is the International Organization for Standards, and the International Electrotechnical Committee (ISO/IEC). This is the path under which networking management is defined.
`joint-iso-ccitt(2)`	The administration authority for this branch is shared between CCITT and ISO/IEC.

Ultimately, every path through the naming tree terminates at a *leaf node*. The sequence of labels along the path (starting at the root) is the OID for the object named at the leaf.

OIDs in Practice

The convention for writing object identifiers is called *dot notation*. An OID in dot notation consists of the integers of the OID in sequence with a period (dot) between them. The prefix for the OIDs in the MIB-II is:

```
1.3.6.1.2.1
```

In the next example, the full name of the path is shown beneath the corresponding numerical identifiers in the OID:

```
1  . 3 . 6 . 1      .  2 . 1   . 6 . 7
iso.org.dod.internet.mgmt.mib-2.tcp.tcpAttemptFails
```

Similarly, the prefix for the OIDs in IBM's enterprise-specific MIBs is:

```
1.3.6.1.4.1.6.3
```

Converting Object Identifiers to Text Strings

The NetView for AIX program provides two API routines that convert MIB variable names between their dot-notation format and their textual equivalents. Use the OVmib_read_objid routine to convert a character string to dot notation. This permits you to build, from the textual name of a MIB variable, an oid in the proper form to submit as a parameter to many SNMP API routines. The function prototype for this routine is:

```
int OVmib_read_objid(const char *name,
                     ObjectID   *oid,
                     u_int      *oid_length);
```

The OVmib_read_objid routine performs a lookup function to convert the input string to dot notation. If the input string begins with a period, the routine converts the string as coded. If the input string does not begin with a period, the routine will attach the following default prefixes:

- .iso.org.dod.internet.mgmt.mib-2
- .iso.org.dod.internet.private.enterprises

Use the OVmib_get_objid_name routine to convert an oid in dot notation to its textual name. This permits you to build, from an oid returned by an API routine, a text string suitable for messages and log files. The function prototype for this routine is:

```
const char OVmib_get_objid_name(ObjectID   *oid,
                                u_int      *oid_length);
```

An example program showing how to use these routines is provided in /usr/OV/prg_samples/nvsnmp_app/name_to_oid.c. Please refer to the *NetView for AIX Programmer's Reference* or the man pages for more information about these routines.

Extended MIBs

Many agents support extended MIBs, which define objects that are not included in standard MIBs. Your application can query an object from an extended MIB exactly as it would query a MIB-II object. Users should work with the proper registration authorities when defining MIB extensions.

Data Representation

Information is exchanged between SNMP processes using the Basic Encoding Rules (BER) defined for the Abstract Syntax Notation (ASN.1). ASN.1 is a very rich data description language; gaining a full understanding of it is a formidable task. The SNMP API takes care of the details of ASN.1 encoding and decoding, so you do not have to deal directly with ASN.1 or the Basic Encoding Rules.

SNMP uses a few simple ASN.1 data types. The following list describes the base data types that SNMP communication uses.

Type	**Description**
INTEGER	A simple type consisting of positive and negative whole numbers, including zero, and of arbitrary size up to 32 bits. However, some objects restrict INTEGER to a range.

OCTET STRING	A simple type taking zero or more octets, each octet being an ordered sequence of eight bits. The value of any octet in the string is unrestricted.
OBJECT IDENTIFIER	An array of integers (unsigned longs). Each integer represents one element of the object identifier.
COUNTER	A non-negative integer that calculates change and increases until it reaches a maximum value, then wraps around and starts increasing again from zero.
GAUGE	A type representing a non-negative integer, which may increase or decrease, but which latches at a maximum value.
TIMETICKS	A type representing a non-negative integer that counts the time in hundredths of a second since some event.
IPADDRESS	A type representing a 32-bit Internet address. It is represented as an OCTET STRING of length 4, in network byte-order. When this ASN.1 type is encoded using the ASN.1 basic encoding rules, only the primitive encoding form shall be used.
OPAQUE	A type representing an arbitrarily-coded ASN.1 string, which has been coded into an OCTET STRING using the basic encoding rules.

The NetView for AIX SNMP API

The NetView for AIX SNMP API is based on Revision 1.1 of the CMU SNMP Library, and offers the following features:

- Blocking and non-blocking function calls
- Support for automatic retransmission
- Support for manual retransmission
- Easy, predictable memory management
- Location transparency for proxies
- The ability to send and receive traps
- Registration of applications to receive specific traps

Blocking and Nonblocking Operation

Your application can make SNMP requests in either of two ways: blocking or non-blocking.

Blocking

When your application issues a blocking SNMP request, it is suspended until either the response arrives or a time-out error occurs on the session. Blocking operation is appropriate for an application that is not event-driven and needs the response information before further processing is warranted.

Nonblocking

When your application issues a non-blocking SNMP request, it can continue other processing while the request is being serviced. The SNMP API function returns as soon as the SNMP interface determines its validity. When the response arrives, your application should call the OVsnmpRead() routine. Then the response is automatically processed by a callback routine that you specified in the non-blocking request. Non-blocking operation is appropriate for an application that is event-driven or can proceed with further processing while waiting for the response.

Retransmission Support

SNMP uses the User Datagram Protocol (UDP) at the transport layer. UDP provides a simple but unreliable transport. It is possible for a message, or part of a message, to get lost in transmission and never arrive at its destination. Services that use UDP, such as SNMP, may require that some messages be retransmitted.

If your application uses blocking requests, the SNMP library provides retransmission based on the values established when the session is first opened. See the OVsnmpOpen() man page for details.

If your application uses non-blocking requests, the NetView for AIX SNMP API provides two ways to manage retransmission:

- Automatic retransmission using the X Extensions
- Manual retransmission

Automatic Retransmission

The SNMP API library includes extended support for event-driven X-based applications. When you use this feature, X and the SNMP library manage all message retransmission, using the XtAppMainLoop() routine.

Manual Retransmission

Applications that are event-driven, but not X-based, make non-blocking requests. These applications can manage their retransmission requirements using the system select() function and two specialized SNMP functions provided for that purpose: OVsnmpDoRetry() and OVsnmpGetRetryInfo(). Refer to the man pages for more information on using these routines.

Memory Management

The NetView for AIX SNMP API uses two simple rules for memory management:

- When you pass a data structure into a library function, it is consumed. The memory it occupies is deallocated by the library.
- When you obtain a data structure from a library function, you must deallocate the associated memory.

Sometimes, a data structure obtained by one function call is consumed by a later one. For example, the call to open a session creates an OVsnmpSession data structure, which is later consumed by the OVsnmpClose call.

Occasionally, you must provide data to fill in a structure that the SNMP library has allocated. For instance, you might provide a list of variables to a request message. Such data must always be dynamically allocated. Otherwise, a failure will occur when the library attempts to deallocate statically allocated memory.

You must ensure that the memory allocated by the SNMP library or by your application is later deallocated. Neglecting to do so can result in a memory leak, which gradually consumes resources until a failure occurs. Use the **free_pdu** flag to overcome this problem.

Note: When a request times out, the callback function is invoked with the type parameter equal to **SNMP_ERR_NORESPONSE**. The library will automatically free the request PDU. No memory deallocation is required.

Location Transparency

The NetView for AIX SNMP implementation provides special support for applications that communicate with an agent through a proxy. Such an application can address a message directly to the true (foreign) agent; the underlying implementation will determine which host is the proxy and route the message accordingly.

The information used to recognize proxy agents resides in the **/usr/OV/conf/ovsnmp.conf** file. This file must be configured by the administrator of the manager.

For Further Reading

The following documents contain useful information that is beyond the scope of this guide.

- *RFC 1155: Structure and Identification of Management Information for TCP/IP-based Internets.* K. McCloghrie and M. T. Rose, (May 1990). Contains MIB object definitions (Obsoletes RFC 1065).
- *RFC 1157: A Simple Network Management Protocol.* J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin, (May 1990). Defines SNMP (Obsoletes RFC 1098).
- *RFC 1187: Bulk Table Retrieval with the SNMP.* K. McCloghrie, M. T. Rose, and C. Davin, (October 1990).
- *RFC 1212: Concise MIB Definitions.* K. McCloghrie and M. T. Rose, (March 1991). Describes the format for creating MIB object files.
- *RFC 1213: Management Information Base Network Management of TCP/IP Base Internets: MIB-II.* K. McCloghrie and M. T. Rose, eds., (March 1991). Defines MIB-II (Obsoletes RFC 1158; most current edition as of the printing of this guide).
- *RFC 1215: Convention for Defining Traps for Use with the SNMP.* M. T. Rose, ed. (March 1991).
- *The Simple Book*. M. T. Rose, (Prentice Hall, 1991). Introduces management of TCP/IP networks using the SNMP.

Chapter 18. Using SNMP API Functions and Data Structures

This chapter describes the basic functions and data structures you will encounter in the NetView for AIX SNMP API. This includes the following items:

- SNMP functions
- The X extensions to the SNMP API for event-driven X-based applications
- Key data structures in the SNMP API

SNMP Functions

There are twenty-five SNMP functions, in five categories. Table 28 describes these functions:

Table 28 (Page 1 of 2). The SNMP API Functions

Category	Function Name	Description
Session Management	`OVsnmpOpen` `OVsnmpXOpen`	Establishes an active SNMP session. Use OVsnmpXOpen in an event-driven X application.
	`OVsnmpTrapOpen` `OVsnmpXTrapOpen`	Opens a session with the SNMP trap daemon, trapd, to receive SNMP traps. Use OVsnmpXTrapOpen in an event-driven X application.
	`nvSnmpTrapOpenFilter` `nvSnmpXTrapOpenFilter`	Sets up to receive specific SNMP traps as specified by a filter parameter. Use nvSnmpXTrapOpenFilter in an event-driven X application.
	`OVsnmpClose` `OVsnmpXClose`	Terminates an active SNMP session and frees associated resources. Use OVsnmpXClose in an event-driven X application.

Table 28 (Page 2 of 2). The SNMP API Functions

Category	Function Name	Description
Message Setup and Management	`OVsnmpCreatePdu`	Allocates an SNMP Protocol Data Unit (PDU) data structure. A PDU contains an SNMP message.
	`OVsnmpAddNullVarBind` `OVsnmpAddTypedVarBind`	Allocates and initializes an OVsnmpVarBind data structure, which is then attached to a PDU. Used in Get, Get Next, and Set operations.
	`OVsnmpFixPdu`	If, in a list of variables, one or more variables cause a request to fail (for any reason), OVsnmpFixPdu can be used to strip the offending variable(s) from the list. The result is a list that can be used to retry the request.
	`OVsnmpFreePdu`	Deallocates a PDU structure and recovers associated resources.
	`OVsnmpErrString`	Converts SNMP error values to text strings.
Communication	`OVsnmpSend` `OVsnmpXSend`	Sends an SNMP message in non-blocking mode. Unless an error occurs, resources associated with the PDU are deallocated with this call. Use OVsnmpXSend in an event-driven X-based application to achieve automatic nonblocking retransmission.
	`OVsnmpBlockingSend`	Sends an SNMP message in blocking mode. Unless an error occurs, resources associated with the PDU are deallocated with this call.
	`OVsnmpRecv`	Receives a message on a specific SNMP session. Does not use a callback.
	`OVsnmpRead`	Receives messages on all active SNMP sessions. Returns information through the callback function.
Manual Retransmission	`OVsnmpGetRetryInfo`	Gets retransmission information on pending SNMP requests.
	`OVsnmpDoRetry`	Retransmits a pending SNMP request.
Table Retrieval	`nvSnmpBlockingGetTable`	Retrieves a MIB variable that is a table. Uses blocking mode.
	`nvSnmpGetTable` `nvSnmpXGetTable`	Retrieves a MIB variable that is a table. Uses nonblocking (event-driven) mode.
	`nvSnmpGetTableElement`	Retrieves a single table element from a table retrieved from a MIB.

Coding Models

This section illustrates the different coding models that can be used with this API:

- Blocking applications
- Nonblocking applications
  - X-based approach
  - Other event-driven approaches

Example programs provide detailed illustration of the manager's operation in each type of communication. The example programs are in the /usr/OV/prg_samples/ovsnmp_app and /usr/OV/prg_samples/nvsnmp_app directories.

Blocking Model

A hypothetical SNMP blocking transaction might use SNMP functions in the following sequence:

Table 29. Sequence of SNMP Function Calls in Blocking Model

Manager	Agent
	Listen for Requests
`OVsnmpOpen()`	
`OVsnmpCreatePdu()`	
`OVsnmpAddVarBind()`	
`OVsnmpBlockingSend()`	Receive request PDU
	Generate response PDU
	Send response PDU
`OVsnmpFixPdu()`	
`OVsnmpClose()`	

This coding model would be normal for an application that does not use the X library and is not otherwise event-driven. For example, this model is appropriate for a command-line application that makes one SNMP request at a time, provides feedback to the user, and waits for the next user request.

In the blocking model, it is not necessary to use select() or other techniques to detect a response. The OVsnmpBlockingSend routine does not return to the calling program until it returns either a response pdu or a time-out message.

If your OVsnmpBlockingSend call times out, returning SNMP_ERR_RESPONSE, you do not need to call OVsnmpFreePdu, but you should call OVsnmpClose to clean up the socket descriptor, session structure, etc.

Nonblocking Model

Nonblocking approaches differ significantly from the blocking approach. There are two nonblocking models: one for X-based applications, another for other event-driven applications. For either case, you provide a pointer to a function to handle the nonblocking arrival of responses. This kind of function is known as a callback function. When a response arrives for your application the callback is invoked to manage the incoming PDU. Exactly how this happens depends on whether your application is X-based.

X-Based Approach: The NetView for AIX SNMP API library has extended functions to support X-based applications. The X environment manages these operations transparently, invoking your callback whenever a response arrives for your application. If you use this approach, you do not need to:

- Issue calls to read or receive responses
- Manage retransmission of lost messages

The coding model for an application that uses the X extensions uses the same logical sequence of calls as those used by a non-X-based nonblocking application, but they are encapsulated in the X callbacks for SNMP requests and responses.

Other Nonblocking Approaches: A hypothetical SNMP nonblocking transaction might employ SNMP functions in the following sequence:

Table 30. Sequence of SNMP Function Calls in Nonblocking Model

Manager	**Agent**
	Listen for Requests
`OVsnmpOpen()`	
`OVsnmpCreatePdu()`	
`OVsnmpAddVarBind()`	
`OVsnmpSend()`	
`OVsnmpGetRetryInfo()`, `select()`, and `OVsnmpDoRetry()` (as necessary)	Read/receive request PDU
	Generate response PDU
	Send response PDU
`OVsnmpRead()` or `OVsnmpRecv()`	
`OVsnmpFixPdu()`	
`OVsnmpClose()`	

After you issue the send call, you must later use select() to determine whether a response has arrived. If so, you must issue a call to the OVsnmpRead() routine to cause the callback to be invoked. Use the OVsnmpRecv() routine to receive the PDU directly without invoking the callback. If the manager is interacting with only one agent, you can use the receive call. If the manager is interacting with multiple agents, you should use the read call. This coding model is appropriate for an event-driven application that does not use the X environment.

Note: You cannot combine the X coding models presented in this section with the other coding models. That is, an X application must use the X-extended functions and refrain from calling select(). Conversely, a non-X application must not use the X extensions.

SNMP Data Structures

This section introduces a few key data structures and their purposes and relationships. Specific details about other parameters to SNMP functions are available in the man pages.

Header Files

The header files define many data structures that are part of the NetView for AIX SNMP API. Table 31 describes the contents of the OVsnmp header files.

Table 31. Contents of OVsnmp Header Files

File Name	Description
/usr/OV/include/OV/OVsnmp.h	Main include file for NetView for AIX SNMP applications. This header file includes OVsnmpApi.h. Also contains protocol definitions, error definitions, and trap type definitions.
/usr/OV/include/OV/OVsnmpApi.h	Main function declarations, structure definitions, and control definitions.
/usr/OV/include/OV/OVsnmpAsn1.h	ASN.1 type definitions for the API. These are the ASN.1 types that are supported by the NetView for AIX SNMP API.
/usr/OV/include/OV/OVsnmpClnt.h	More function declarations.
/usr/OV/include/OV/OVsnmpImpl.h	More ASN.1 types. These are for convenience.
/usr/OV/include/OV/OVsnmpXfns.h	Function declarations for the X11 extensions in the NetView for AIX SNMP API.

Refer to the SNMP man pages for information on compiling and linking your application.

Data Structures

Table 32 describes key SNMP data structures.

Table 32. SNMP Data Structures

Data Structure Name	Created By	Freed By	Description
OVsnmpSession	OVsnmpOpen OVsnmpTrapOpen nvSnmpTrapOpenFilter	OVsnmpClose	Identifies a particular SNMP communication session. Used as an input parameter to several other functions.
OVsnmpPdu	OVsnmpCreatePdu	OVsnmpFreePdu	Container for an SNMP message. Includes the message data and information about the type of message (Get, Get Next, Set, or Trap).
OVsnmpVarBind	OVsnmpAddVarBind	OVsnmpFreePdu	Elements in a linked list of variables. Each element of the list includes an object identifier for the variable and the variable's value. Part of the `OVsnmpPdu` structure.
OVsnmpVal	OVsnmpAddVarBind	OVsnmpFreePdu	A union that can contain an integer, a string, or an object identifier. Part of the `OVsnmpVarBind` structure.

You will use two structures in particular: the OVsnmpSession structure, and the OVsnmpPdu structure (including its substructures).

The OVsnmpSession Structure

The OVsnmpSession structure is allocated by a call to the OVsnmpOpen() routine and is an input parameter to several other functions. It has the following fields, as shown in Table 33:

Table 33 (Page 1 of 2). OVsnmpSession Structure Fields

Type	Name	Description
u_char	*community	The agent's community name for get requests. Defaults to **public**. When the session is closed, this memory is deallocated.
u_int	community_len	The number of bytes in the get community name.
int	sock_fd	The socket file descriptor for the session; used to establish the read mask for calls to select(3).

Table 33 (Page 2 of 2). OVsnmpSession Structure Fields

Type	**Name**	**Description**
u_short	session_flags	A bitmask for information and control. The following values are defined: IS_PROXIED_FOR: if set, notes that destination node is being proxied for by an anonymous node. The library sets this value appropriately. FREE_PDU: defaults to **set**. If set, the send functions deallocate memory for PDUs. RECV_TRAPS: if set, requests that traps be accepted by this session. If the session is not created by OVsnmpTrapOpen(), this allows you to receive traps on your local port. Most agents send traps to port 162, which is reserved for SNMP, and monitored by the SNMP trap daemon. Use OVsnmpTrapOpen() to receive traps on port 162. The default for this flag is **Unset**.
(void)()	(*callback) ()	Points to the callback function to be invoked when a response returns from a call to OVsnmpSend() or OVsnmpXSend(). These two send functions are nonblocking, and the callback function is to be invoked when the corresponding response arrives.
void	*callback_data	Points to data the callback function is to receive (note the syntax of the callback invocation in "The Callback Function" on page 295). The memory is not accessed by the library.
u_char	* setCommunity	The agent's community name for set requests. When the session is closed, this memory is deallocated.
u_int	community_len	The number of bytes in the set community name.
int32	nvSnmpBind	Internal use only; do not modify.
octet	*TNRdecodeCnf	Internal use only; do not modify.
OVeCDNode	*TNRderegSieve	Internal use only; do not modify.

The Callback Function

The intent of your application determines the structure and purpose of the callback function in the previous table. If you use the nonblocking calls in the SNMP API, the callback function defines the interaction of your application and the remote peer.

The callback function is automatically invoked when your application uses OVsnmpRead() to obtain the response to a nonblocking send request. In X applications, the callback is invoked automatically when the response arrives. The call syntax is shown in the following example:

```
callback (type, session, pdu, callback_data)
```

If the response is obtained with the OVsnmpRecv() routine, the callback does not occur.

Table 34 describes the parameters your callback function must define. These input parameters are passed to the callback function. "The OVsnmpPdu Structure" on page 296 describes these parameters in more detail.

Table 34. Callback Function Parameters

Type	Name	Description
int	type	One of the following SNMP message types, or `SNMP_ERR_NO_RESPONSE` if a time out occurs: `GET_REQ_MSG`, `GETNEXT_REQ_MSG`, `SET_REQ_MSG`, or `TRAP_REQ_MSG` (if the session is receiving traps). `type` indicates the type of PDU that generated the response.
OVsnmpSession	*session	Identifies the session with which the incoming PDU is associated.
OVsnmpPdu	*pdu	This is the PDU that was received. This parameter will be NULL if a time out occurred.
void	*callback_data	Application-specific data, which is specified as a value passed into the OVsnmpOpen() call.

The OVsnmpPdu Structure

This data structure contains all the data specific to a particular SNMP message. It is created in one of two ways:

- When preparing to send a request, a call is made to the OVsnmpCreatePdu() routine.
- When a response or trap is received.

The SNMP send functions normally deallocate the memory associated with input PDU structures. To deallocate the PDU associated with a response or trap, use the OVsnmpFreePdu() routine.

Table 35 describes each of the elements.

Table 35 (Page 1 of 2). Elements of the OVsnmpPdu Data Structure

Type	Field Name	Description
ipaddr	address	The IP address of the agent. Your application never needs to set this value; it is ignored in a call to a send function.
int	command	The specific SNMP command. Must be one of the following: `GET_REQ_MSG`, `GETNEXT_REQ_MSG`, `SET_REQ_MSG`, `TRAP_REQ_MSG`, or `GET_RSP_MSG`.
int	request_id	An identifier assigned by the SNMP API. This value must not be modified by your application.
int	error_status	When a response PDU is received, this variable contains a positive value if an error occurred on the request. If no error occurred, the value is zero. This value is ignored in a call to a send function.

Table 35 (Page 2 of 2). Elements of the OVsnmpPdu Data Structure

Type	Field Name	Description
int	error_index	An index into the variable list. If error_status shows that an error occurred, then error_index indicates which element of the list caused the error.
OVsnmpVarBind	variables	This is a pointer to a linked list of variables, each element of which is an OVsnmpVarBind data structure. Memory referenced by this pointer is deallocated by a call to OVsnmpFreePdu(), OVsnmpFixPdu(), or any send function.
Note: The following fields are specific to trap PDUs and are not valid in any other PDUs.		
ObjectID	*enterprise	A vendor-specific identifier that uniquely identifies the type of device sending the trap. Defaults (on send) to **1.3.6.1.4.1.2.6.3.1**, a generic object identifier for Netview specific traps.
u_int	enterprise_length	The number of elements in the object ID. The default is 10 elements.
u_long	agent_addr	The IP address of the agent that sent the trap. Defaults (on send) to the IP address the PDU is sent on.
int	generic_type	One of the SNMP-defined types of traps. See RFC 1157 for details.
int	specific_type	When generic_trap indicates that an enterprise-specific trap has occurred, the value of specific_trap contains the specific trap. Otherwise, the value is meaningless.
u_long	time	The time (in tenths-of-seconds) since the agent was last activated. The default is the length of time that the application has been running.
Note: The following fields are part of the embedded OVsnmpVarBind data structure(s).		
OVsnmpVarBind*	next_variable	Points to the next element in the variable list. `NULL` in last element.
ObjectID	*oid	The object identifier (object ID) for this variable.
u_int	oid_length	The number of elements in the object ID for this variable.
u_char	type	The ASN.1 type of the variable.
OVsnmpVal	val	A union containing a long integer, a string, or an object ID. The active field is indicated by type.
u_int	val_len	The number of elements in the value of the variable. Note that this may not equal the number of bytes in `val`; for example, an object ID consists of an array of long integers.

The OVsnmpConfEntry Structure

This is the primary data structure used by the SNMP configuration routines.

Table 36. Elements of the OVsnmpConfEntry Data Structure

Type	Field Name	Description
char	*name	The name of the target node. It can be a hostname or alias, an IP address, or a proxy name.
char	*community	Community name for SNMP commands.
char	*setCommunity	Community name for SNMP set commands.
char	*proxy	Name of proxy to use.
int	timeout	Length of time, in tenths of seconds, to wait before retrying a request. The value must be greater than 0.
int	retry	The number of times to retry a request before concluding that the node in inoperable. The value must be greater than or equal to 0.
int	pollInterval	The IP status polling interval, in seconds.
unsigned short	remotePort	The SNMP port number on the target node.

The SNMP configuration routines also use the OVsnmpConfCntl, OVsnmpConfWcList, and OVsnmpConfDest data structures. See the OVsnmpIntro man page for information about these structures.

SNMP API Coding Examples

This section contains two sample programs that use the NetView for AIX SNMP API. The first example issues a nonblocking get request and waits for a reply. The second example illustrates the use of the table retrieval routines in both blocking and nonblocking modes.

Any application that uses the OVsnmp API must link to /usr/OV/lib/libovsnmp.a or /usr/OV/lib/libnvsnmp.a.

SNMP Nonblocking Get Sample

```
#include <OVsnmp.h>
#include <stdio.h>
#include <sys/select.h>
#include "/usr/include/malloc.h"
#include "snmpdemo.h"

#define NUMARGS 3

static struct oid_info
{
   char     *name;
   ObjectID oid[MAX_SUBID_LEN];
   int      oid_len;
   u_char   type;
} oids[7] = { {"sysDescr",      {1,3,6,1,2,1,1,1,0}, 9, ASN_OCTET_STR},
              {"sysObjectID",   {1,3,6,1,2,1,1,2,0}, 9, ASN_OBJECT_ID},
              {"sysUpTime",     {1,3,6,1,2,1,1,3,0}, 9, TIMETICKS},
              {"sysContact",    {1,3,6,1,2,1,1,4,0}, 9, ASN_OCTET_STR},
              {"sysName",       {1,3,6,1,2,1,1,5,0}, 9, ASN_OCTET_STR},
              {"sysLocation",   {1,3,6,1,2,1,1,6,0}, 9, ASN_OCTET_STR},
              {"sysServices",   {1,3,6,1,2,1,1,7,0}, 9, INTEGER}
            };

static int num_oids = 7;

struct nonBlockingInfo
{
   int       waiting;
   int       error;
   OVsnmpPdu *response;
};

static char            *testName;
OVsnmpSession          *session;
struct nonBlockingInfo info;

void           printVariable(OVsnmpVarBind *var);
void           nonBlockingGet();
void           pduCallback(int type, OVsnmpSession *session,
                           OVsnmpPdu *response, void *userData);
void           closeSession(OVsnmpSession *session);

/***************************************************************************/
/*****************************  main  **************************************/
/***************************************************************************/

int main(int argc, char **argv)
{
   const char            *community;
   const char            *destination;

   testName = argv[0];

   if (argc != NUMARGS)
   {
      printf("%s: Incorrect arg count %d\n", testName, argc);
      printf("usage: %s hostname community\n", testName);
      exit(1);
   }

   destination   = argv[1];
   community     = argv[2];

   session = OVsnmpOpen(community, destination,
                        SNMP_USE_DEFAULT_RETRIES,
                        SNMP_USE_DEFAULT_INTERVAL,
                        SNMP_USE_DEFAULT_LOCAL_PORT,
                        SNMP_USE_DEFAULT_REMOTE_PORT,
                        pduCallback, &info);
   if (session == NULL)
   {
      printf("Unable to initialize SNMP session: ");
      printf("%s\n", OVsnmpErrString(OVsnmpErrno));
      exit(1);
   }

   info.waiting  = 1;
   info.response = NULL;
   info.error    = 0;

   nonBlockingGet(community, destination);
   closeSession(session);
}                                                        /* end of main */

/***************************************************************************/
/*************************  nonBlockingGet  ****************************/
/***************************************************************************/

void nonBlockingGet()
{
   OVsnmpPdu          *requestGet;
   OVsnmpVarBind      *varp;
   int                NBresponse,
                      numFDs;
   struct fd_set      readFDs;
   struct timeval     timeVal;

   int i, count;

   session->session_flags &=  FREE_PDU;

   if ((requestGet = OVsnmpCreatePdu(GET_REQ_MSG)) == NULL)
   {
      printf("Unable to create PDU. %s.\n", OVsnmpErrString(OVsnmpErrno));
      exit(1);
   }

   for (i = 0; i < num_oids; i++)
   {
      varp = OVsnmpAddNullVarBind(requestGet, oids[i].oid, oids[i].oid_len);
      if (varp == NULL)
      {
         printf("%s.\n", OVsnmpErrString(OVsnmpErrno));
         OVsnmpFreePdu(requestGet);
         closeSession(session);
      }
   }

   NBresponse = OVsnmpSend(session, requestGet);
   if (NBresponse == -1)
   {
      printf("Error sending request. %s.\n", OVsnmpErrString(OVsnmpErrno));
      OVsnmpFreePdu(requestGet);
      closeSession(session);
   }

   while (info.waiting == 1)
   {
      numFDs = OVsnmpGetRetryInfo(&readFDs, &timeVal);
      count = select(numFDs, &readFDs, NULL, NULL, &timeVal);
      if (count < 0)
      {
         printf("select");
         closeSession(session);
      }
      else if (count > 0)
      {
         OVsnmpRead(&readFDs);
      }
      else
      {
         OVsnmpDoRetry();
      }
   }
   OVsnmpFreePdu(requestGet);
}

/****************************************************************************/
/*************************  pduCallback  ***********************************/
/****************************************************************************/

void pduCallback(int type, OVsnmpSession *session, OVsnmpPdu *request,
                 void* cData)
{
   OVsnmpVarBind          *varp;
   struct nonBlockingInfo *data =  (struct nonBlockingInfo*) cData;

   data->waiting = 0;

   if (type == SNMP_ERR_NO_RESPONSE)
   {
      printf("%s.\n", OVsnmpErrString(type));
      OVsnmpFreePdu(request);
      return;
   }

   if (request->error_status == SNMP_ERR_NOERROR)
   {
      for (varp = request->variables; varp !=NULL;
           varp = varp->next_variable)
      {
         printVariable(varp);
      }
   }
   else
      printf("(callback) %s.\n", OVsnmpErrString(OVsnmpErrno));
   OVsnmpFreePdu(request);
   return;
}
/****************************************************************************/
/*************************  closeSession  **********************************/
/****************************************************************************/

void closeSession(OVsnmpSession *session)
{
   if (OVsnmpClose(session) < 0)
   {
      printf("Error closing session! %s.\n", OVsnmpErrString(OVsnmpErrno));
      exit(1);
   }
   else
   {
      printf("\n\nSession closed successfully.\n");
      exit(0);
   }
}

/**************************************************************************/
/***********************  printVariable  ********************************/
/**************************************************************************/

void printVariable(OVsnmpVarBind *var)
{
   int buflen;
   char buf[1028];

   buflen = sizeof(buf);
   sprint_by_type(buf, buflen, var, VAL_ONLY);
   printf("%s\n", buf);
}
```

SNMP Table Retrieval Sample

```
#include <stddef.h>
#include <sys/types.h>
#include <sys/select.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <OV/OVsnmp.h>
#include <OV/OVsnmpClnt.h>
#include <OV/OVsnmpXfns.h>
#include "snmpdemo.h"

#define NUMARGS 3

struct info
{
   int block;
   int cont; /* abbreviation for continue */
   int rows;
   int columns;
};

static char *programName;

void          printVariable(OVsnmpVarBind *var,
                            int          tableFormat);
void          callback(int type,
                       OVsnmpSession *session,
                       OVsnmpPdu *response,
                       void *userData);

/**************************************************************************/
/****************************  main  *************************************/
/**************************************************************************/
```

```
int main(int argc, char **argv)
{
   OVsnmpVarBind     *ptr;
   OVsnmpPdu         *pduPtr;
   const char        *destination,
                     *community;
   char              *table;
   int               oidLen = 9;
                     /************************/
                     /* ObjectID for IfEntry */
                     /************************/
   static ObjectID   oid1[MAX_SUBID_LEN] = {1,3,6,1,2,1,2,2,1},
                     oid2[MAX_SUBID_LEN] = {1,3,6,1,2,1,3,1,1};
                     /************************/
                     /* ObjectID for AtEntry */
                     /************************/
   OVsnmpSession     *session;
   int               response,
                     rows, columns,
                     numFDs, count;
   struct fd_set     readFDs;
   struct timeval    timeVal;
   struct info       callbackdata;

   if (argc < NUMARGS)
   {
      printf("Usage: %s <hostname> <community> \n", programName);
      exit(1);
   }

   programName = argv[0];
   destination = argv[1];
   community   = argv[2];

   /***************************************************************/
   /* Open a session to be passed to the nvSnmp-GetTable() calls.    */
   /* This is where you can provide a callback function and callback */
   /* data to be used.                                               */
   /***************************************************************/
   session = OVsnmpOpen(community,
                        destination,
                        SNMP_USE_DEFAULT_RETRIES,
                        SNMP_USE_DEFAULT_INTERVAL,
                        SNMP_USE_DEFAULT_LOCAL_PORT,
                        SNMP_USE_DEFAULT_REMOTE_PORT,
                        callback,
                        &callbackdata);

   if (session == NULL)
   {
      printf("OVsnmpOpen: %s\n", OVsnmpErrString(OVsnmpErrno)); exit(1);
   }

   /******************************************************/
   /* block = 1  means that we are doing a blocking call.    */
   /* This is used for control within the callback function. */
   /******************************************************/
   callbackdata.block = 1;
   /*******************************************************/
   /* This is a blocking call.  It will not return until a */
   /* timeout has occurred or the data has been received.  */
   /*******************************************************/
   pduPtr = nvSnmpBlockingGetTable(session, oid1, oidLen, &(callbackdata.rows),
                                   &(callbackdata.columns));
   printf("# of Rows = %d\n# of Columns =  %d\n",
          callbackdata.rows, callbackdata.columns);

   /**********************************************************/
   /* block = 0 means that we are doing a nonblocking call.    */
   /* This is used for control within the callback function.   */
   /**********************************************************/
   callbackdata.block = 0;

   /***********************************************************/
   /* This is a nonblocking call.  It will return immediately.  */
   /* The requested PDU will be available within the callback   */
   /* specified when the session was established.               */
   /***********************************************************/
   response = nvSnmpGetTable(session, oid2, oidLen, &(callbackdata.rows),
                             &(callbackdata.columns));

   /**********************************************************/
   /* callbackdata.cont is set within the callback function.  */
   /* It is used to determine when all of the table has been  */
   /* retrieved by the nonblocking call.                      */
   /**********************************************************/
   while (!callbackdata.cont)
   {
      numFDs = OVsnmpGetRetryInfo(&readFDs, &timeVal);
      count = select(numFDs, &readFDs, NULL, NULL, &timeVal);
      if (count < 0)
      {
         printf("Error in select. Check errno."); exit(1);
      }
      else if (count > 0)
      {
         OVsnmpRead(&readFDs);
      }
      else
      {
         printf("The select timed out - retrying via OVsnmpDoRetry().");
         OVsnmpDoRetry();
      }
   } /* end of while (!callbackdata)  */
}

/*********************************************************************/
/*************************** callback() ********************************/
/*********************************************************************/
void callback(int type, OVsnmpSession *session, OVsnmpPdu *response,
              void *userData)
{
   OVsnmpVarBind   *ptr, *elementPtr;
   struct info     *data = (struct info*) userData;
   int             i, j, tableFormat;
   printf("\nCallback called.\n");
   if (type == SNMP_ERR_NO_RESPONSE)
   {
      printf("No Response Returned (within user callback)\n");
      exit(1);
   }
   printf("Data from within callback:\n");
   printf("   # of Rows = %d\n   # of Columns =  %d\n",
          data->rows, data->columns);
   elementPtr = nvSnmpGetTableElement(response, data->rows, 0, 0);
   printf("   Element[0,0]: ");
   tableFormat = 1;
   printVariable(elementPtr, tableFormat);
   printf("\n");

   elementPtr = nvSnmpGetTableElement(response, data->rows, 1, 2);
   printf("   Element[1,2]: ");
   tableFormat = 1;
   printVariable(elementPtr, tableFormat);
   printf("\n");

   printf("response->command = %d\n", response->command);
   printf("response->request_id = %d\n", response->request_id);
   printf("response->error_status = %d\n", response->error_status);
   if (data->columns < 4)
      tableFormat = 1;
   else
      tableFormat = 0;

   for (i = 0; i < data->rows; i++)
   {
      for (j = 0; j < data->columns; j++)
      {
         elementPtr = nvSnmpGetTableElement(response, data->rows, i, j);
         printVariable(elementPtr, tableFormat);
/*
         if (tableFormat)
            printf("   ");
*/
      }
      printf("\n");
   }

   /*********************************************************/
   /* If this callback was called due to a nonblocking call,  */
   /* set cont to TRUE so that calling program knows all      */
   /* data has been received.                                 */
   /*********************************************************/
   if (data->block == 0)
      data->cont = 1;
}
/***************************************************************************/
/************************  printVariable  **********************************/
/***************************************************************************/

static void printVariable(OVsnmpVarBind *var, int tableFormat)
{
   int     buflen, match;
   char    buf[1028];

   buflen = sizeof(buf);
   sprint_by_type(buf, buflen, var, VAL_ONLY);
   printf("%s", buf);
   if (tableFormat)
      printf("   ");
   else
      printf("\n");
}
```

Chapter 19. Filtering Network Events

Events are a critical part of a modern network management system. The occurrence of events dictates much of the processing done in such a system. Because of this, you may want to control the processing of events in certain ways:

- Limit how often your application receives certain events.
- Limit how certain events are sent to remote manager stations.
- Enable applications to receive events only from certain network nodes.
- Ignore events generated during a certain part of the day.

The NetView for AIX program enables you to do all this, and more, through the use of *event filters*. Filtering network events can eliminate unnecessary or repetitive processing of frequent events and reduce the number of alerts sent to the NetView interface.

The NetView for AIX *filter editor* is part of the NetView for AIX graphical interface. This editor enables the user to define, modify, and delete filtering rules for use by applications. Refer to the *NetView for AIX User's Guide for Beginners* for information about using the filter editor before developing an application that will use event filters.

Creating Filters

Depending on the purpose of your application, you may want to limit the events it receives. The NetView for AIX filter API routines enable you to define, modify, delete, or retrieve filters for use within your application. The following routines are provided:

- nvFilterDefine
- nvFilterDelete
- nvFilterErrorMsg
- nvFilterFreeNameList
- nvFilterGet
- nvFilterGetNameList

An application that will use these routines must include the nvFilter library, `/usr/OV/lib/libnvfilter.a`.

The FilterNode Structure

An include file is provided that contains the function headers and a filter structure, named `FilterNode`, that will be used as a parameter to several of these routines. In your code, include the file as follows:

```
#include <nvFilter.h>
```

The structure has the following definition:

```
struct FilterNode
{
   char *FilterName;
   char *FilterDescription;
   struct FilterNode *Next;
}
```

Parameter	Description
FilterName	A pointer to the name of a filtering rule.
FilterDescription	A pointer to the description of a filtering rule. This field is optional.
Next	A pointer to the next FilterNode. This field is used only by the nvGetFilterNameList routine.

NetView for AIX Filter API Routines

This section describes the syntax and parameters of the routines in the NetView for AIX Filter API. Complete details on each routine are provided in the *NetView for AIX Programmer's Reference* and in the man pages

Defining a Filter Rule

Use the nvFilterDefine() routine to create a new filtering rule or to update an existing rule. It has the following function prototype:

```
int nvFilterDefine(struct FilterNode *Filter, char *FileName,
                   char *FilterStr, int Update);
```

Parameter	Description
Filter	A pointer to a filter structure containing the name of the filtering rule and an optional description of the rule.
FileName	The path and name of the filter file.
FilterStr	The filtering rule.
Update	Zero (0) - do not update; one (1) - update contents if the rule exists.
Returns	Zero (0) if no error occurred; if an error occurred, one of the return codes listed in Table 37 on page 310.

Deleting Filter Rules

Use the nvFilterDelete() routine to remove a filtering rule from the filter file. It has the following function prototype:

```
int nvFilterDelete(char *RuleName, char *FileName);
```

Parameter	Description
RuleName	The name of the filtering rule to be deleted.
FileName	The path and name of the filter file to be deleted.
Returns	An error if the filtering rule does not exist.

Retrieving a Filter Rule

Use the nvFilterGet() routine to retrieve the contents of a filtering rule. It has the following function prototype:

```
int nvFilterGet(struct FilterNode *Filter, char *FileName,
                char *Buffer, int  *BufLen, int Expand);
```

Parameter	Description
Filter	A pointer to a filter structure containing the name of the filtering rule. If a description exists for the specified rule, it is returned in the `FilterDescription` field.
FileName	The path and name of the filter file.
Buffer	A pointer to the memory location where the rule will be written. If this field is NULL, the length of the filtering rule will be returned in the `BufLen` parameter.
BufLen	A pointer to the size of the buffer. If the buffer is too small, an error is returned and `BufLen` is changed to reflect the actual size of the filtering rule.
Expand	Zero (0) - do not expand references to other filtering rules and hostnames; one (1) - expand the references.
Returns	Zero (0) if no error occurred; if an error occurred, one of the return codes listed in Table 37 on page 310.

If there are two or more filters defined with the same name, this routine will return the first filter found by searching from the top of the specified file.

To determine the size of the buffer required, you can call the nvFilterGet routine once with *Buffer* set to NULL, and then use the value returned in BufLen in a second call to retrieve the rule.

Space is allocated for the rule description on each call to the nvFilterGet routine. Your application is responsible for deallocating this memory between calls to this routine. These techniques are illustrated in “Example of Filter API Routines” on page 311.

Listing Filter Rules

Use the nvFilterGetNameList() routine to retrieve a list of all the filtering rule names and descriptions in the filter file. It has the following function prototype:

```
int nvFilterGetNameList(char *FileName, struct FilterNode **FilterList);
```

Parameter	Description
FileName	The path and name of the filter file.
FilterList	The address of a pointer to a `FilterNode` structure. Contains the address of the first node in the link list upon returning.
Returns	Zero (0) if no error occurred; if an error occurred, one of the return codes listed in Table 37 on page 310.

This routine returns a list of unique filter names. If the specified file contains more than one filter with a given name, only the first filter encountered will be included in the list.

Freeing Filter Rule Storage

Use the nvFilterFreeNameList() routine to free the memory allocated during the creation of the list of filtering rule names. It has the following function prototype:

```
void nvFilterFreeNameList(struct FilterNode *FilterList);
```

Parameter	Description
FilterList	A pointer to the head of the filter list.
Returns	There is no return parameter.

Reading Error Messages

Use the nvFilterErrorMsg() routine to retrieve the error message that corresponds to a Filter API return code. It has the following function prototype:

```
char *nvFilterErrorMsg(int Retcode);
```

Parameter	Description
Retcode	The return code from the Filter API call.

When you pass a return code from a Filter API routine to the nvFilterErrorMsg routine, it returns a message from the list in Table 38 on page 311.

NetView for AIX Filter API Return Codes

When an NetView for AIX Filter API routine is called, it passes a return code to the calling program to indicate the result of the call. A return code of zero (0) indicates successful completion of the call; otherwise, one of the following return codes is passed on completion of the call:

Table 37. Return Codes from Filter API Calls

Value	Name
1	NVFILTER_FILE_NOT_FOUND
2	NVFILTER_FILE_ACCESS_ERROR
3	NVFILTER_MEMORY_ACCESS_ERROR
4	NVFILTER_FILTERNAME_NOT_FOUND
5	NVFILTER_INSUFFICIENT_SPACE
6	NVFILTER_DUPLICATE_FILTERNAME
7	NVFILTER_FILTER_FILE_EMPTY
8	NVFILTER_INCORRECT_FILTER_FILE_FORMAT
20	NVFILTER_HOSTNAME_RESOLUTION_ERROR
21	NVFILTER_FILTER_RESOLUTION_ERROR
22	NVFILTER_FILTER_REFERENCE_ERROR
23	NVFILTER_TIME_FORMAT_ERROR
24	NVFILTER_MAX_BUFFERSIZE_EXCEEDED

The man pages for these routines list the return codes that can be returned by each routine. The return code names listed in Table 37 are for use in your code; "Example of Filter API Routines" on page 311 illustrates checking a return code for a particular value from the table. Table 38 on page 311 lists the messages that are returned when you pass one of these return codes to the nvFilterErrorMsg routine:

Table 38. Error Messages from Filter API Calls

Value	Message
0	Filter file access request completed successfully.
1	The requested filter file was not found.
2	Error occurred while attempting to access the filter file.
3	Memory access error.
4	The specified filter was not found in the filter file.
5	Insufficient space for requested filter.
6	The filtername already exists in the filter file.
7	The filter file is empty.
8	The filter file is not in the required format.
20	Error occurred while attempting to resolve a hostname.
21	Error occurred while attempting to resolve a reference to another filter.
22	Filter reference format error.
23	Time format error.
24	Filter exceeds filter buffer size.
Other	Unknown Filter API error.

Example of Filter API Routines

The following example illustrates the use of the nvFilterGetNameList and nvFilterGet routines. In this example, the list of filters stored in `/tmp/filterFile` is retrieved, then each filter from the list is retrieved and printed. Note the careful management of memory: the free command is used to free the storage for the current filter description before each call to the nvFilterGet routine, and the nvFilterFreeNameList routine is used to free the memory allocated by the nvFilterGetNameList routine. Notice the two calls to the nvFilterGet routine, as described in “Retrieving a Filter Rule” on page 309.

```
#include <stdio.h>
#include <string.h>
#include "/usr/include/malloc.h"
#include <nvFilter.h>

void main()
{
   struct FilterNode   *filterList, *current;
   char                *buffer;
   int                 rc, bufferLen = 0, j = 0, expand = 0;
   char                *filterFile = "/tmp/filterFile";

   rc = nvFilterGetNameList(filterFile, &filterList);
   if (rc < 0) {
      printf("%d: %s\n", rc, nvFilterErrorMsg(rc));
   }
   for (current = filterList; current !=NULL; current = current->Next)
   {
      free(current->FilterDescription);
      /*  call nvFilterGet once to get required bufferLen value  */
      rc = nvFilterGet(current, filterFile, NULL, &bufferLen, expand)
      if (rc != NVFILTER_INSUFFICIENT_SPACE) {
         printf("%d: %s\n", rc, nvFilterErrorMsg(rc));
      }

      buffer = (char *) malloc(bufferLen * sizeof(char));
      free(current->FilterDescription);
      /*  call nvFilterGet again to get the rule  */
      rc = nvFilterGet(current, filterFile, buffer, &bufferLen, expand);
      if (rc != 0) {
         printf("%d: %s\n", rc, nvFilterErrorMsg(rc));
      }

      printf("Buffer %d: %s\n", j, buffer);
      j++;
      free(buffer);
   }
   nvFilterFreeNameList(filterList);
}
```

Using Filters

The NetView for AIX filter API routines enable you to manipulate filter files and rules, but they do not register your application to receive filtered events. The registration process you use depends on which network management protocol you use.

If your application uses the NetView for AIX SNMP API, you can register for filtered events using two routines that are part of that API.

Step 1. Create the filter rule, using the NetView for AIX filter editor or the Filter API routines described in this chapter.

Step 2. Use the nvSnmpTrapOpenFilter routine or the nvSnmpXTrapOpenFilter routine to register your application using your filter. These routines are described in Chapter 18, “Using SNMP API Functions and Data Structures” on page 289.

If your application uses the XMP API, you can control its registration using the routines described in this section. To register for filtered events when using XMP, follow these steps:

Step 1. Create the filter rule, using the NetView for AIX filter editor or the Filter API routines described in this chapter.

Step 2. Use the OVeFilterAttr() routine to convert the filter rule to an XOM data structure.

Step 3. Use the OVeRegister() routine to register your application using the data structures you have just created.

The OVeFilterAttr() and OVeRegister() routines are described next.

NetView for AIX Event Registration Routines

Use the following convenience routines to control your application's event registration when using XMP:

- OVeFilterAttr()
- OVeRegister()
- OVeDeregister()

Many of the parameters used by these routines are XOM data structures. XOM is an API that is used with XMP to manage the complex data structures it requires. XOM is described in Chapter 12, “Using the XOM API” on page 189.

For detailed information about files that must be included when calling these routines, see the man page for the routine you will use.

Before using these routines to register an event filter for your application, create the filter with either the NetView for AIX filter editor or the Filter API routines. After creating the filter rules by one of these means, you must convert them into XOM data structures before you can use them with XMP. Use the OVeFilterAttr() routine to perform this conversion.

Creating the XOM Structure for Your Filter

Use the OVeFilterAttr() routine to convert an existing filter rule into an XOM data structure. It has the following function prototype:

```
int OVeFilterAttr( OM_workspace workspace,
                   char *filter_string,
                   OM_private_object *out_filter_attribute,
                   char **err_ptr,
                   OM_return_code *om_error);
```

The *workspace* parameter specifies an XMP workspace in which the XOM object will be created. See “XMP Workspaces” on page 224 for more information on XMP workspaces. The OVeFilterAttr() routine accepts a filter string as input and returns an XOM structure that can be passed to the OVeRegister() routine to register the filter. Refer to the man page for more information about the parameters and syntax of the input filter string.

Registering Your Filter

After creating a filter structure with OVeFilterAttr(), use the OVeRegister() routine to register your application to receive events. It has the following function prototype:

```
int OVeRegister( OM_private_object session, OM_workspace workspace,
                 OVeConvRegNode *node_list,
                 OM_private_object filter_attribute,
                 OVeConvConfirm **confirm_list,
                 OM_return_code *om_error);
```

Refer to the man page for more information on this routine and its parameters.

Creating Filters on Remote Manager Stations

If your network includes more than one manager station running the NetView for AIX program, you can establish event filters on each management station. An application on one management station can register to receive certain events from another management station by setting up a filter on the remote station. To establish a remote filter, pass a list of nodes to the OVeRegister() routine in the `node_list` parameter.

Cancelling Application Registration

Use the OVeDeregister() routine to cancel an application's registration to receive events. Specify the nodes, including the local node and any remote nodes, at which filters will be deleted, and the filters that are to be deleted. Filters and locations are identified in a list of objects returned from the OVeRegister() call.

The OVeDeregister() routine has the following function prototype:

```
int OVeDeregister( OM_private_object session, OM_workspace workspace,
                 OVeConvDeRegNode *filter_list,
                 OVeConvConfirm **confirm_list,
                 OM_return_code *om_error);
```

The *filter_list* parameter is a pointer to a linked list of destination addresses, object classes, and instances defining the filter objects to be deleted. The elements of this list were returned when the application registered these filters by calling the OVeRegister() routine. Refer to the man page for more information about this routine and its parameters.

Chapter 20. Using the General Topology Manager

You can use the NetView for AIX program, with no modifications, to manage networks that use the IP and SNMP protocols. You can also use the NetView for AIX program to manage networks that use other protocols, by creating applications to pass information to the NetView for AIX program in the required format. The NetView for AIX General Topology Manager (GTM) accepts information about devices that use other protocols, stores this information in a database, and presents the information to the user. This chapter introduces the GTM and the Open Topology MIB, which is used to store topology data about non-IP networks. Chapter 21, “Communicating with the General Topology Manager” on page 343 explains how to set up applications to use the GTM to perform these tasks.

Introducing the General Topology Manager

This section describes the benefits of using the General Topology Manager, the components that make up the GTM, and some key terms that are used in describing topology elements.

Benefits of Using the General Topology Manager

Although it is possible to construct a separate application to gather and display information about networks that use other protocols, there are many advantages, both for you as a developer and for your user, to integrating this information with the NetView for AIX program. This section lists the benefits of this approach to both developers and users.

Benefits for the Developer

Using the NetView for AIX GTM to integrate your network's information with the NetView for AIX program provides these benefits to you as an application developer:

- Enables you to present topology information to your user through the xxmap application, so that you do not need to build a separate application to display topology information.
- Enables you to integrate the gathering of your protocol data with the NetView for AIX discovery process.
- Correlates non-IP data with IP data, so that duplicate object information is not stored in the database.
- Provides your user the benefits described below without your writing any applications.

Benefits for Your User

Using the NetView for AIX GTM to integrate and present information about other networks helps your user understand how networks are connected. The GTM performs these functions:

- Integrates non-IP networks and devices with IP and other protocols that integrate their information with the NetView for AIX program.
- Correlates the non-IP data with IP data so that multiprotocol nodes are presented properly.

- Allows your user to specify whether status is to be propagated across protocols or only within a protocol.
- Allows your user to switch views of a device running multiple protocols
- Provides event-card integration (pressing the highlight button for a non-IP event will display the submap where the node is displayed within that protocol, just as it does for an IP event). The highlight button is not supported for non-IP events that are not received through the NetView for AIX GTM.

For more information on presenting non-IP topology information to your user, see "Presenting Topology Information to the User" on page 356.

Components of the NetView for AIX General Topology Manager

The programs that you write to provide information about your non-IP network will interact with two NetView for AIX daemons: noniptopod and gtmd.

The noniptopod Daemon

The **/usr/OV/bin/noniptopod** (noniptopod) daemon registers to receive traps from netmon (through ovesmd) that indicate the discovery of a new node that has an IP address. The noniptopod daemon receives the new node's IP address from the trap. As described in "Discovering Nodes" on page 343, it determines whether the node supports any of a list of non-IP protocols. If it finds that the node supports a non-IP protocol, noniptopod issues a command to begin gathering topology data from that node.

The gtmd Daemon

The **/usr/OV/bin/gtmd** (gtmd) daemon receives topology information that describes the attributes of devices on a non-IP network. This daemon stores topology information in its database and stores object-related information in the NetView for AIX object database for use by display applications.

The gtmd and noniptopod daemons are, by default, not started when the NetView for AIX program is started. You can use SMIT to configure them and indicate that they should be started by the nv6000 command.

The Open Topology MIB

The data structures you will use to describe open-topology elements are defined in the NetView for AIX Open Topology MIB. Note that this MIB is not a loadable MIB like one that represents a managed object; it is a set of data structures to be used in representing general topology data. The MIB also defines actions that can be performed on these topology elements, and a set of status values. All these MIB elements are described in the remainder of this chapter.

The Open Topology MIB is stored in the /usr/OV/snmp_mibs/drafts/ibm-nv6ktopo.mib file.

The xxmap Application

The NetView for AIX General Topology Manager uses the xxmap application to display topology information.

The gtmdump Utility

The gtmdump utility is a troubleshooting tool for displaying the contents of the GTM database and monitoring GTM trap processing. Current values in the Open Topology MIB can be dumped to a file for review by NetView for AIX product support.

This command can be found in the *Programmer's Reference*.

Understanding Key Terms

The following terms, many of which are borrowed from graph theory, are defined to describe the syntax of generic topology objects and connections. In addition to the graph theory model, a layered connectivity model is used to show the relationship between the higher and lower layers of the communications network. These terms will be used throughout this chapter:

Vertex A vertex is a point in some space. A set of vertices is connected by any number of arcs to form a graph. A vertex contains logical or physical interfaces to a network. Logical interfaces are protocols, such as APPN* or IP. Physical interfaces are hardware adapters, such as token ring, Ethernet, or FDDI.

Arc An arc represents connectivity between vertices or graphs acting as vertices. An example of an arc is a connection between two IP hosts or connectivity via token ring. An arc is the representation of a connection that is independent of either end point. You can define an arc to represent the end-to-end delivery of a function or application, while using simple and underlying connections to represent the actual path over which the function is provided.

Graph A graph is a representation of a set of vertices and the connections between them. It represents a physical network, such as a token ring or an Ethernet, or a logical network, such as IP or APPN. In addition to representing physical or logical resources, a graph can be used to group resources in a network based on any criteria, for example, all the network elements in a single building. By showing a graph as a vertex, a topology application can treat the graph as a single unit for topology display.

In addition to representing a logical or physical network, a graph can represent a computer node. Each of the computer's components is represented as a vertex with the accompanying connections. This sort of graph is called a *box graph.*

Member The members of a graph are all the vertices and other graphs that are contained within the graph.

Member arc
An arc is a member arc of a graph if it is completely contained within the graph.

Underlying arc
An underlying arc is an arc that represents a connection that provides the lower-level connectivity used by another arc.

Simple connection
A simple connection represents a connection as seen from one of its end points. A simple connection can include the number of octets sent

on the connection, operability of the connection as seen from an end point, or other information that is specific to an end point.

Underlying connection
Lower-layer connections used by a higher layer are called underlying connections. For example, a connection between two IP hosts uses the physical connection for its transport. The physical connections used to transport data between the hosts are the underlying connections. The layered connectivity model is used to represent the dependencies between connections in a communication stack.

Service access point
A service access point (SAP) is the mechanism by which a lower-layer network element provides access to its services for higher-layer elements. For example, an IP host might use a LAN for its physical transport layer. The vertex representing the LAN station provides a SAP to be used by the host.

A vertex can use only one SAP. The SAP used by the vertex represents a lower-level protocol, and each vertex can use only one lower-level protocol. A resource can provide services to many other resources through one or more SAPs. To represent these relationships, SAP usage is represented as a table.

The basic concepts of vertex and simple connection convey the basic connectivity information about any resource. A graph provides a way to group resources or represent a topology. The use of layered connectivity enables you to model the relationships between physical and logical networks, or between the networks and the functions they provide.

More information about each of these key terms is provided in the following discussion of how they are represented in the tables of the Topology MIB.

Open Topology MIB Tables and Groups

The NetView for AIX Open Topology MIB is defined by a set of twelve tables that contain object and relationship information. The tables are divided into four groups:

- Vertex Group
  - Vertex Table
  - SAP Table
- Arc Group
  - Arc Table
  - Underlying Arc Table
- Simple Connection Group
  - Simple Connection Table
  - Underlying Connection Table
- Graph Group
  - Graph Table
  - Members Table
  - Member Arcs Table
  - Attached Arcs Table

    – Additional Members Table
    – Additional Graph Table

Each group consists of a single primary table and several secondary tables. The primary table represents an object. The secondary tables represent relationships between the objects. A group consists of all the tables and variables that describe all the resources of a single class.

The primary table contains one entry for each resource of the type represented by the table. The secondary tables provide additional information about resources in a group. They provide a mechanism for unlimited-length lists and a place for information that applies to a subset of resources in the primary tables.

Each entry in a secondary table is associated with one row in its primary table. In the secondary tables, there can be multiple entries for a single row in the primary table. To enable easy association of the information in the secondary tables with a single row in a primary table, the index variables for the secondary tables will be the index variables for the primary table. In each of the following table descriptions, the index variables will be shown in bold type.

Creation and Deletion of Group Entries

When an entry is created in a primary table of a group, the associated entries in the secondary tables can also be created. Additional entries can be added to the secondary tables for a resource at times other than resource creation. When an entry is deleted from a primary table, all associated information is deleted at the same time from the secondary tables. Associated information can be deleted without deleting all information about a resource.

In the following group descriptions, the first table is always the primary table. All other tables in the group are secondary tables.

The Vertex Group

The vertex group contains all the information necessary to represent a vertex. It consists of the following tables:

- Vertex table
- Service access point table

The Vertex Table

A vertex represents a resource that contains an interface within a managed system. An interface provides access to a network or service. A vertex will either be a physical resource such a token ring, Ethernet or FDDI, or a logical resource such as IP or APPN.

A vertex contains two variables for naming: *vertexProtocol* and *vertexName*. The variable *vertexProtocol* is the prefix for the vertex name. The variable *vertexName* is the name. The *vertexProtocol* and *vertexName* variables are the index variables for the vertex table. They are used as part of the index fields for the service access point (SAP) and simple connection tables.

Each vertex has operability characteristics. The vertex table contains the state management variables defined in section “State and Status Information” on page 335.

The information described above for vertices is resource-independent. It is the basic information needed to define any vertex; it is not sufficient to represent a complete vertex. The variables *vertexManagementExtension* and *vertexManagementAddr* are provided to store additional resource-specific information.

Graphs, arcs, and simple connections will refer to vertices. If a referenced vertex is not represented by the agent that represents a referencing graph, arc, or simple connection, the local agent must create a vertex to contain the name of the referenced vertex, because all references between tables are local. The *vertexMine* field indicates whether a vertex is represented by the agent that owns the vertex table. If *vertexMine* has a value of **MINE**, then the vertex is represented by the local SNMP agent, and all the fields in the vertex are valid. If *vertexMine* is equal to **NOT_MINE**, the vertex is represented by another agent, and the entry in the local table contains only the information that the local agent knows about the vertex. For a vertex with *vertexMine* equal to **NOT_MINE**, the operational state field has no meaning.

Note: The gtmd daemon and the xxmap application do not use the *vertexMine* field. It is for the use and control of discovery or management applications.

The variable *vertexLocation* is a readable string that can be used to store the vertex location. This variable is not displayed by the xxmap application.

Table 39. The Vertex Table

Attribute	**Type**
vertexProtocol	nvotVertexProtocolType
vertexName	char *
vertexMine	nvotOwnerType
vertexLocation	char *
vertexManagementExtension	nvotOctetString
vertexManagementAddr	nvotOctetString
vertexOperationalState	nvotOperationalStateType
vertexUnknownStatus	nvotUnknownStatusType
vertexAvailabilityStatus	nvotAvailabilityStatusType
vertexAlarmStatus	nvotAlarmStatusType
vertexLabel	char *
vertexIcon	char *

The Service Access Point (SAP) Table

While vertices are the means to represent communication entities or interfaces across various protocol layers, the SAP object class is the representation of the logical relationship between two vertices inside a computer. When a communication entity in a given protocol layer makes use of the services of a lower-layer entity through a service point, the vertex representing the higher-layer entity *uses a SAP provided by* a vertex representing an entity in the lower layer. Likewise, a vertex representing an entity in the lower layer *provides SAPs for use by* vertices representing entities in the higher layer. Moreover, a given interface or communication entity may provide its services to more than one entity in a higher layer at the same time.

The variables *sapVertexProtocol* and *sapVertexName* refer to the vertex either using or providing the SAP.

The *sapServiceType* indicates whether the vertex is using or providing the SAP.

The variables *sapProtocol* and *sapAddress* define the SAP itself. Refer to the man pages for the nvotCreateUsingSap and nvotCreateProvidingSap routines for a complete explanation of the use of these variables.

Table 40. The Service Access Point Table

Attribute	**Type**
sapVertexProtocol	nvotVertexProtocolType
sapVertexName	char *
sapServiceType	nvotServiceType
sapProtocol	nvotVertexProtocolType
sapAddress	char *

The Simple Connection Group

The simple connection group contains all information necessary to represent a simple connection. It consists of the following tables:

- Simple connection table
- Underlying connection table

The Simple Connection Table

A simple connection represents the information about a connection that is known only to the endpoint, such as total bytes received or total bytes received with an error. It is the point of attachment of an arc in a vertex or graph, even if the arc does not exist. In this case, the simple connection represents an intention of connection between two endpoints. The partner of a simple connection is the vertex or graph that this endpoint would like to connect.

A simpleConnection has three index variables: *localEndpointProtocol*, *localEndpointName* and *simpleConnIndexId*. The *localEndpointProtocol* and *localEndpointName* fields identify the vertex or graph for which this is endpoint-specific information about one of its connections. The *localEndpointProtocol* and *localEndpointName* fields contain the same values as the index variables for the local connection endpoint in the graph or vertex tables. The variable *simpleConnIndexId* is an instance variable that identifies all simple connections with the same *localEndpointProtocol* and *localEndpointName* values.

Associated with the index field, there is a name binding field that determines if the simple connection is associated to a vertex or a graph. The defined values are:

- SIMPLE_CONN_VERTEX_NAME_BINDING
- SIMPLE_CONN_GRAPH_NAME_BINDING

This field is not part of the simple connection attributes; It is defined in the simple connection structure. The index variables for simple connection form part of the index variables of the underlying connection table.

The variable *simpleConnName* is the human-readable label for the simple connection to be used on a topology display.

A simple connection identifies the partner at the other end of the connection with the variables *connectionPartnerProtocol* and *connectionPartnerName*. These variables have identical values to the index variables in the vertex or graph table for the connection partner. If the local vertex does not know the vertex at the other end of the connection, this information is omitted. In order to determine if the partner is a vertex or a graph, the variable *nameBinding* should be filled with one of the values below:

- VERTEX_NAME_BINDING
- GRAPH_NAME_BINDING

The variables *simpleConnManagementExtension* and *simpleConnManagmentAddr* are pointers to detailed information about the simple connection.

The variable *simpleConnIcon* identifies the icon used to represent this simple connection.

A simple connection also has operability characteristics. See section “State and Status Information” on page 335 for further details.

Table 41. The Simple Connection Table

Attribute	**Type**
localEndpointProtocol	nvotProtocolType
localEndpointName	char *
simpleConnIndexId	int
simpleConnName	char *
nameBinding	nvotNameBindingType
connectionPartnerProtocol	nvotProtocolType
connectionPartnerName	char *
simpleConnManagementExtension	nvotOctetString
simpleConnManagementAddr	nvotOctetString
simpleConnIcon	char *
simpleConnOperationalState	nvotOperationalStateType
simpleConnUnknownStatus	nvotUnknownStatusType
simpleConnAvailabilityStatus	nvotAvailabilityStatusType
simpleConnAlarmStatus	nvotAlarmStatusType

The Underlying Connection Table

Some connections consist of other connections. These other connections are called underlying connections. There are two types of underlying connections: parallel and serial. A *parallel* underlying connection is the set of simple connections that run in parallel between the same endpoints. A *serial* underlying connection is an ordered sequence of simple connections between two endpoints. Serial underlying connections are thought of as running one after the other to form the connection. Between any two endpoints, there can be parallel underlying connections, serial underlying connections, or a combination of parallel and serial underlying connections.

This table identifies the underlying connections of a simple connection. The underlying connections of a simple connection are simple connections. The index variables for the underlying connection table are *ulcEndpointProtocol*, *ulcEndpointName*, *ulcEndpointId*, *nameBinding*, *uconnEndpointProtocol*, *uconnEndpointName* and *uconnSimpleConnId*.

The variables *ulcEndpointProtocol*,*ulcEndpointName* and *ulcEndpointId* are equivalent to the *localEndpointProtocol*, *localEndpointName* and *simpleConnIndexId* variables of the simpleConnection table. They identify the simple connection that is underlying of another simple connection. Associated with the index field, there is a name binding field that determines if this simple connection is associated to a vertex or a graph. The defined values are:

- SIMPLE_CONN_VERTEX_NAME_BINDING
- SIMPLE_CONN_GRAPH_NAME_BINDING

The variables *nameBinding*, *uconnEndpointProtocol*, *uconnEndpointName*, and *uconnSimpleConnId* are also equivalent to the index variables of the simple connection table. They identify the simple connection which has the underlying connection described by this table.

Each individual underlying connection may be parallel or serial. The variable *underlyingConnectionKind* defines whether the individual underlying connection is parallel or serial. It has two values: **PARALLEL** and **SERIAL**. When the underlying connection is **SERIAL**, a sequence of lower-level links run between the two endpoints of a connection. The variables *nextSerialEndpointProtocol*, *nextSerialEndpointName* and *nextSerialSimpleConnId* are the instance of the next serial connection. By combining these variables with *ulcEndpointProtocol*, *ulcEndpointName* and *ulcEndpointId* until the next serial variables are found with ***don't care*** values, you can discover the connections that make up the serial connection order. The variable *underlyingConnectionKind* cannot be changed. If the kind of an individual underlying connection changes, all underlying connections for the using simple connection must be deleted and a new set of underlying simple connections added.

Table 42 (Page 1 of 2). The Underlying Connection Table

Attribute	Type
ulcEndpointProtocol	nvotProtocolType
ulcEndpointName	char *
ulcEndpointId	int
underlyingConnectionKind	nvotUnderlyingKindType
nameBinding	nvotNameBindingType
uconnEndpointProtocol	nvotProtocolType
uconnEndpointName	char *
uconnSimpleConnId	int
nextSerialNameBinding	nvotNameBindingType
nextSerialEndpointProtocol	nvotProtocolType

Table 42 (Page 2 of 2). The Underlying Connection Table

Attribute	Type
nextSerialEndpointName	char *
nextSerialSimpleConnId	int
ulcIcon	char *
ulcLabel	char *

The Arc Group

The arc group contains all the information necessary to represent an arc. It consists of the following tables:

- Arc table
- Underlying arc table

The Arc Table

An arc shows a connection between two vertices or graphs. An arc may represent a real resource, such as a SNA session or a TCP connection, or it can be a *user-defined* arc. A user-defined arc provides a method to aggregate multiple real resources into a single entity. A user-defined arc provides an aggregation function for arcs just as a graph does for a topology. A user-defined arc could represent the connection between a client display and the server machine that provides an application that is viewed on the client display. This connection could involve several physical connections, which might use different protocols. In this case, these physical connections could be represented as serial underlying connections, with SAPs used to identify the relationship between the connections and the arc.

The arc identifies its endpoints with the pairs of fields, *aEndpointProtocol*, *aEndpointName*, and *zEndpointProtocol*, *zEndpointName*. Each endpoint pair contains the value of the index fields of the endpoint in either the graph or vertex table.

The index fields for the arc table are *aEndpointProtocol*, *aEndpointName*, *zEndpointProtocol*, *zEndpointName*, and *arcIndexId*. There can be many arcs between the same endpoints; *arcIndexId* serves as an instance identifier to distinguish between arcs with the same endpoints. Associated with the index field, there is a name binding field that determines, for each endpoint, whether it is a vertex or a graph. The defined values are:

- ARC_VERTEX_VERTEX_NAME_BINDING
- ARC_VERTEX_GRAPH_NAME_BINDING
- ARC_GRAPH_VERTEX_NAME_BINDING
- ARC_GRAPH_GRAPH_NAME_BINDING

This field is not part of the arc attributes; It is defined in the arc structure. The index fields from the arc form part of the index for the underlying arcs table.

The fields *aDetailsIndexId* and *zDetailsIndexId* are the additional index fields needed to build the name of the simple connection for each endpoint of the arc. These fields contain the *simpleConnIndexId* value for each endpoint.

The variables *arcManagementExtension* and *arcManagmentAddr* are detailed information about the arc.

An arc also has operability characteristics. See “State and Status Information” on page 335 for further details. For an arc, the operability characteristics refer to the arc as a whole.

Table 43. The Arc Table

Attribute	**Type**
aEndpointProtocol	nvotProtocolType
aEndpointName	char *
zEndpointProtocol	nvotProtocolType
zEndpointName	char *
arcIndexId	int
aDetailsIndexId	int
zDetailsIndexId	int
arcManagementExtension	nvotOctetString
arcManagementAddr	nvotOctetString
arcOperationalState	nvotOperationalStateType
arcUnknownStatus	nvotUnknownStatusType
arcAvailabilityStatus	nvotAvailabilityStatusType
arcAlarmStatus	nvotAlarmStatusType

The Underlying Arc Table

The underlying arcs table is semantically equivalent to the underlying connection table. The major difference is that the underlying connection table points to simple connections while the underlying arcs table points to arcs.

The index fields for this table are *ulaAendpointProtocol*, *ulaAendpointName*, *ulaZendpointProtocol*, *ulaZendpointName*, *ulaArcIndexId*, *ulaArcIndexId*, *nameBinding*, *uconnAendpointProtocol*, *uconnAendpintName*, *uconnZendpointProtocol*, *uconnZendpointName* and *uconnArcIndexId*.

The first five fields contain the same value as the index fields for the arc table. These fields identify the arc which is using the underlying arc. Associated with the index field, there is a name binding field that determines, for each endpoint, whether it is a vertex or a graph. The defined values are:

- ARC_VERTEX_VERTEX_NAME_BINDING
- ARC_VERTEX_GRAPH_NAME_BINDING
- ARC_GRAPH_VERTEX_NAME_BINDING
- ARC_GRAPH_GRAPH_NAME_BINDING

This field is not part of the underlying arc attributes. It is defined in underlying arc structure.

The variables *nameBinding*, *uconnAendpointProtocol*, *uconnAendpointName*, *uconnZendpointProtocl*, *uconnZendpointName* and *uconnArcIndexId* are also equivalent to the index variables of the arc table. They identify the arc which has the underlying arc described by this table.

Each underlying arc may be either parallel or serial. The variable *underlyingArcKind* defines whether the underlying arc is parallel or serial. It has two values **PARALLEL** and **SERIAL**. When the underlying arc is **SERIAL**, a sequence of lower-level links run between the two endpoints of a connection. The variables *nextSerialAendpointProtocol*, *nextSerialAendpointName*, *nextSerialZendpointProtocol*, *nextSerialAendpointName* and *nextSerialArcIndexId* are the instance of the next serial arc. By combining these variables with *ulaAendpointProtocol*, *ulaAendpointName*, *ulaZendpointProtocol*, *ulaZendpointName* and *ulaArcIndexId* until the next serial variables are found with a ***don't care*** values, you can discover the connections that make up the serial arc order. The variable *underlyingArcKind* can not change. If the kind of an individual underlying arc changes, the complete set of underlying arcs for the using arc must be deleted and a new set created.

Table 44. The Underlying Arc Table

Attribute	Type
ulaAendpointProtocol	nvotProtocolType
ulaAendpointName	char *
ulaZendpointProtocol	nvotProtocolType
ulaZendpointName	char *
ulaArcIndexId	int
underlyingArcKind	nvotUnderlyingKindType
nameBinding	nvotNameBindingType
uconnAendpointProtocol	nvotProtocolType
uconnAendpointName	char *
uconnZendpointProtocol	nvotProtocolType
uconnZendpointName	char *
uconnArcIndexId	int
nextSerialNameBinding	nvotNameBindingType
nextSerialAEndpointProtocol	nvotProtocolType
nextSerialAendpointName	char *
nextSerialZendpointProtocol	nvotProtocolType
nextSerialZendpointName	char *
nextSerialArcIndexId	int
ulaIcon	char *
ulaLabel	char *

The Graph Group

The graph group contains all information necessary to represent a graph. It consists of the following tables:

- Graph table
- Members table
- Member arcs table
- Attached arcs table
- Additional members table
- Additional graph table

The vertex and simple connection groups must be supported by all applications that use this MIB. The arc and graph groups are optional. If an agent implements the graph group it must also implement the arc group.

For each of the groups above, an implementation may need to define resource-specific tables to contain nongeneric information. Each resource-specific table is associated with an implementation of a single group. Each resource-specific table must use the same index variables as the primary table for the group in which it is defined. If there can be multiple entries in the resource specific table for a single resource, an instance identifier must be added to the index variables.

The Graph Table

A Graph is used to represent either a network hierarchy or a box which contains the vertices which are part of topologies. There are two basic types of graphs: the topology-graph and the box-graph.

The topology-graph and the box-graph are different in nature. A topology-graph is typically a connected graph which represents the topology network. The topology-graph represents either a resource-specific topology or a higher-level topology. The higher-level topology is a graph where membership is defined based on management domains or any user defined grouping.

The box-graph represents the vertices contained in a machine. The vertices within a machine can be both physical and logical interfaces. Although GTM and xxmap do not have any restriction, there are typically no arcs between vertices within a box-graph.

Both types of graphs can contain vertices and other graphs, although box-graphs typically contain only vertices. A vertex may be part of many different graphs.

The topology-graph serves two purposes. The first purpose is to depict a resource-specific topology, such as a LAN ring, an APPN network node graph, or the containment of vertices within a single machine. The second purpose is to provide a means to group nodes and other graphs in a network into higher-level graphs. Higher-level graphs are constructed based on user-defined criteria, such as physical location or owning organization, and represent user-defined partitions of a network. Higher-level graphs provide a way to represent partitions of a topology based on management domains, other ways in which the network is managed, or however the user thinks about the network.

When drawing a topology display, a graph is drawn as a vertex. A graph is used to provide the zoom-in, zoom-out function commonly associated with a graphical topology display by providing a mechanism to represent different levels of abstraction about a network. In a topology, this function is used to show the connectivity of a network or a subnetwork. In a box, this function is used to show the placement of different network interfaces within a single box.

The *graphType* field indicates if the table row is a box or a graph. This variable cannot be changed. The value **BOX** indicates that this graph is a box. The value **GRAPH** indicates that this is a topology. The different types of graphs are usually named differently. The topology-graphs could be named by their resource-specific names, such as an IP domain name or a SNA netid. The higher-level graphs (a machine, all the network in a building) are given user-defined names. The box-graphs could be named by a combination of manufacturer, make, model and serial number, if available. Each set of names will have distinct prefixes. The prefix and name fields are *graphProtocol* and *graphName*. These are also the index fields for this table. The index variables are the basis for the index fields for the members, member arcs, attached arcs, additional members, and additional graph tables.

The *layoutAlgorithm* field defines the layout algorithm that the xxmap should use when drawing the graph. The layout algorithms are those used by the NetView for AIX program and the variable cannot be changed. The defined values are:

- NONE_LAYOUT
- POINT_TO_POINT_LAYOUT
- BUS_LAYOUT
- START_LAYOUT
- SPOKED_RING_LAYOUT
- ROWCOL_LAYOUT
- POINT_TO_POINT_RING_LAYOUT
- TREE_LAYOUT

The variable *graphLocation* is a readable string and could be used to store the graph location. This variable is not displayed by xxmap.

The variable *backgroundMap* is a figure that will be displayed as background when this graph is exploded.

The variables *graphManagementExtension* and *graphManagmentAddr* contain detailed information about the graph.

The *isRoot* field indicates whether this graph is the top graph in a hierarchy of graphs. For any hierarchy of graphs only one graph is the root.

The *graphLabel* and *graphIcon* fields should be used only when this graph is an endpoint of an underlying arc. Use these attributes to eefine the label and icon of the graph endpoint.

Table 45. The Graph Table

Attribute	**Type**
graphType	nvotGraphType
graphProtocol	nvotGraphProtocolType
graphName	char *
layoutAlgorithm	nvotLayoutType
userDefinedLayout	char *
graphLocation	char *
backgroundMap	char *
graphManagementExtension	nvotOctetString
graphManagementAddr	nvotOctetString
isRoot	nvotBooleanType
graphLabel	char *
graphIcon	char *

The Members Table

The members table identifies the vertices and graphs which are part of a graph. Each row of the members table is named by five variables: *memberProtocol*, *memberName*, *nameBinding*, *memberComponentProtocol* and *memberComponentName*. The variables *memberProtocol* and *memberName* are the same values that identify the graph in the graph table.

The variables *memberComponentProtocol* and *memberComponentName* are used to identify the vertex or graph which is inside the graph. In order to determine if the member is a vertex or a graph, the variable *nameBinding* should be filled with one of the values below:

- VERTEX_NAME_BINDING
- GRAPH_NAME_BINDING

The variable *memberLabel* is the human readable label that will be displayed by the graphic application. A vertex or a graph could have different labels depending on which graph it is member of.

The variable *memberIcon* defines an icon to represent a vertex or a graph inside an specific graph.

Table 46. The Members Table

Attribute	**Type**
memberProtocol	nvotGraphProtocolType
memberName	char *
nameBinding	nvotNameBindingType
memberComponentProtocol	nvotProtocolType
memberComponentName	char *
memberLabel	char *
memberIcon	char *

The Member Arcs Table

The member arcs table provides a list of all arcs that have both endpoints completely within a graph. For each graph that has arcs within it, there is one table entry associating the arcs with the graph in which they are contained. An arc may be contained within many graphs.

Each row of the members table is named by eight variables: *maGraphProtocol*, *maGraphName*, *nameBinding*, *maAendpointProtocol*, *maAendpointName*, *maZendpointProtocol*, *maZendpointName* and *maArcIndexId*. The variables *maGraphProtocol* and *maGraphName* are the same values that identify the graph in the graph table.

The variables *nameBinding*, *maAendpointProtocol*, *maAendpointName*, *maZendpointProtocol*, *maZendpointName* and *maArcIndexId* are used to identify the arc which is inside the graph. In order to determine which kind of arc is this member arc, the variable *nameBinding* should be filled with one of the values below:

- ARC_VERTEX_VERTEX_NAME_BINDING

- ARC_VERTEX_GRAPH_NAME_BINDING
- ARC_GRAPH_VERTEX_NAME_BINDING
- ARC_GRAPH_GRAPH_NAME_BINDING

The variable *maLabel* is the human-readable label that will be displayed by the graphic application. An arc could have different labels depending on which graph it is member arc of.

The variable *maIcon* defines an icon to represent the arc inside an specific graph.

Table 47. The Member Arcs Table

Attribute	Type
maGraphProtocol	nvotGraphProtocolType
maGraphName	char *
nameBinding	nvotNameBindingType
maAendpointProtocol	nvotProtocolType
maAendpointName	char *
maZendpointProtocol	nvotProtocolType
maZendpointName	char *
maArcIndexId	int
maLabel	char *
maIcon	char *

The Attached Arcs Table

The graph attached arcs table provides a list of all arcs that have an endpoint within the graph. For each graph that has arcs attached, there is one table entry for each associated arc.

A graph attached arc row is named by the graph to which the arc is attached and the arc itself; this serves to associate the attached arc with each graph to which it is attached. The graph name will consist of two fields: *aaGraphProtocol* and *aaGraphName*. These fields contain the corresponding values of the *graphProtocol* and *graphName* fields in the graph table for the graph to which the arc is attached.

The arc name are six fields, which are index in arc table: *nameBinding*, *aaAendpointProtocol*, *aaAendpointName*, *aaZendpointProtocol*, *aaZendpointName* and *aaArcIndexId*.

Table 48 (Page 1 of 2). The Attached Arcs Table

Attribute	Type
aaGraphProtocol	nvotGraphProtocolType
aaGraphName	char *
nameBinding	nvotNameBindingType
aaAendpointProtocol	nvotProtocolType
aaAendpointName	char *
aaZendpointProtocol	nvotProtocolType

Table 48 (Page 2 of 2). The Attached Arcs Table

Attribute	Type
aaZendpointName	char *
aaArcIndexId	int

The Additional Members Table

This table contains optional additional information about a graph member. A graph member has a row in this table when the graph has a layout algorithm of **NONE_LAYOUT**.

This table has index fields: *aMemberProtocol*, *aMemberName*, *nameBinding*, *aMemberComponentName* and *aMemberComponentName*. These variables contain the same values that identify the graph member in the members table.

The fields *xCoordinate* and *yCoordinate* define the location on the display of this member when the layout algorithm is **NONE_LAYOUT**.

The fields *xGrid* and *yGrid*, which define the scale of the icon for this member on the display of the graph, are currently not supported by the NetView for AIX program.

Table 49. The Additional Members Table

Attribute	Type
aMemberProtocol	nvotGraphProtocolType
aMemberName	char *
nameBinding	nvotNameBindingType
aMemberComponentProtocol	nvotProtocolType
aMemberComponentName	char *
xCoordinate	int
yCoordinate	int
xGrid	int
yGrid	int

The Additional Graph Table

This table contains additional information about a graph. To tie the additional information back to the graph it applies to, the index fields for this table contain the same values as those used by graph plus an instance identifier to allow for multiple entries for each graph. The variables *addGraphProtocol* and *addGraphName* contain the same values as the *graphProtocol* and *graphName* for the main graph information. The variables *addGraphIndexId* is a unique instance identifier for each entry for the same graph.

For any layout algorithm that has a root or center member, the variables *graphRootProtocol* and *graphRootName* contain the value of the instance identifier of the root vertex in the members table. Explicit variables are not needed for the protocol and name since *addGraphProtocol* and *addGraphName* are also part of the indices into the members table.

The variable *graphDescr1* contains descriptive information about the graph. This variable is not displayed by xxmap.

The variables *graphDescrX* and *graphDescrY* are the x and y coordinates to be used to place the information in *graphDesc1* on the display of a graph.

As a root graph is not member of any other graph and the label and icon information of a graph are contained in members table, the variables *rootGraphLabel* and *rootGraphIcon* identify the label and the icon of a root graph.

Table 50. The Additional Graph Table

Attribute	**Type**
addGraphProtocol	nvotGraphProtocolType
addGraphName	char *
addGraphIndexId	int
graphRootProtocol	nvotGraphProtocolType
graphRootName	char *
graphDesc1	char *
graphDescrX	int
graphDescrY	int
rootGraphLabel	char *
rootGraphIcon	char *

Open Topology MIB Traps

The Open Topology MIB defines a set of traps to be used to communicate topology changes to network management programs. These traps can be used to inform managers of:

- Newly-discovered vertices or connections
- Deletion of network resources
- Changes of status
- Changes in variable values

List of Traps

Table 51 lists the traps defined in the Open Topology MIB and supported by the NetView for AIX program. The traps are organized by the table to which they apply. The names are self-explanatory; see the MIB for the syntax, hexadecimal code, and description of each trap type.

Table 51 (Page 1 of 2). Open Topology MIB Traps

Table	**Trap Name**
Vertex	newVertex deletedVertex vertexStateChange vertexVariableChange
SAP	newSAPTrap deletedSAP

Table 51 (Page 2 of 2). Open Topology MIB Traps

Table	Trap Name
Simple Connection	newSimpleConnection deletedSimpleConnection simpleConnectionStateChange simpleConnectionVariableChange
Underlying Connection	newUnderlyingConnection deletedUnderlyingConnection
Arc	newArc deletedArc arcStateChange arcVariableChange
Underlying Arc	newUnderlyingArc deletedUnderlyingArc
Graph	newGraph deletedGraph graphVariableChange
Member Arc	newMemberArc deletedMemberArc
Members	newMemberTrap deletedMember
Additional Member Information	newMemberInformation memberInformationVariableChange
Additional Graph Information	newAdditionalGraphInformation additionalGraphInformationVariableChange

The preceding table does not include all the traps defined in the Open Topology MIB. There are some tables, traps, and variables that are defined in the MIB but not currently used by the NetView for AIX program. These items are defined in the MIB for architectural integrity and possible future use. Table 52 lists the items that are not supported. Do not use the following traps or variables to send topology information to the NetView for AIX program:

Table 52 (Page 1 of 2). Restrictions on NetView for AIX Topology MIB Traps and Tables

MIB Table	Restrictions/Comments
Vertex	Do not use vertexMine variable.
SAP	Do not use sapVariableChange trap.
Simple Connection	Do not use the following variables in the simpleConnectionVariableChange trap: • connectionPartnerProtocol • connectionPartnerName
Underlying Connection	Do not use underlyingConnectionKind variable. Do not use nextSerialUlcIndexId variable.
Arc	Only use change of aDetailsIndexId and zDetailsIndexId variables from -1 to a value. No other changes to aDetailsIndexId or zDetailsIndexId are supported.
Underlying Arc	Do not use underlyingArcKind variable. Do not use nextSerialUlaIndexId variable.

Table 52 (Page 2 of 2). Restrictions on NetView for AIX Topology MIB Traps and Tables

MIB Table	Restrictions/Comments
Graph	Do not use userdefinedLayout variable. Do not use the following variables in the graphVariableChange trap: • graphType • layoutAlgorithm
Member Arc	Do not use memberArcVariablechange traps.
Graph-Attached Arcs	Table is not supported.
Members	Do not use memberVariablechange traps.
Additional Member	Do not use xGrid variable. Do not use yGrid variable.
Additional Graph	Do not use graphDesc1 variable. Do not use graphDescrX variable. Do not use graphDescrY variable.

Another restriction not related to a specific table involves the names given to particular protocols. An item in the NetView for AIX object database cannot have a list structure in its name field. This restricts a particular protocol to have only one unique name per entity. Care must be taken in the use of SAPs so that this restriction is not violated. When an entity, for example a vertex, provides a SAP, that SAP is also treated as a name for the vertex object. Thus, that OVw object cannot have two different names with the same protocol.

Limitations on Changing Variable Values

The following table describes, for each table in the NetView for AIX Open Topology MIB, which attributes do not support the variable value change function. Index variables are mandatory and do not support changes.

Table 53 (Page 1 of 2). NetView for AIX Open Topology MIB Limitations

Table	Attribute
Vertex	No restrictions
Graph/Box	graphType layoutAlgorithm
Arc	Only support change of aDetailsIndexId and zDetailsIndexId attributes from -1 to a value and from a value to -1. No other changes to aDetailsIndexId and zDetailsIndexId are supported.
SAP	Do not support changes
Simple connection	connectionPartnerProtocol connectionPartnerName
Underlying connection	Do not support changes
Underlying arc	Do not support changes
Members	No restrictions
Member arc	No restrictions

Table 53 (Page 2 of 2). NetView for AIX Open Topology MIB Limitations

Table	Attribute
Attached arc	Do not support changes
Additional member	No restrictions
Additional graph	No restrictions

State and Status Information

The state and status variables defined for the Open Topology MIB are based on *ISO 10164-2, State Management*. This section will summarize the relevant variables and the expected values. There are more states and statuses in *ISO 10164-2* than those described here, but only those states and statuses described here are used by the NetView for AIX Open Topology MIB.

Operational State

Operational state indicates whether a resource is physically installed and working. It has two distinct states:

Enabled The resource is fully or partially operable and available for use.

Disabled The resource is totally inoperable and unavailable to provide service to the user.

In addition to the operational state, three status fields are defined. These status fields provide additional information about the state of a resource. The status fields used are unknown status, availability status, and alarm status.

Unknown Status

Unknown status indicates whether the state of the resource is unknown. Unknown status is true when an agent cannot reflect the current state of a resource. When this field is true, the operational state, availability status, and alarm status may not be accurate and will reflect the last known values.

Availability Status

Availability status provides further qualification about the state of the resource. It is set valued and has the value of the empty set when none of the conditions apply. Availability status has the following values that indicate restrictions on operational state:

In test The resource is undergoing a test procedure.

Failed The resource has some fault which prevents it from operating. Operational state is disabled.

Power off The resource is not turned on. Operational state is disabled.

Off line The resource requires a routine operation be performed to place it on line and make it available for use. The operation may be manual or automatic. Operational state is disabled.

Off duty The resource is inactive based on a schedule. Operational state is enabled or disabled.

Dependency	The resource cannot operate because some other resource on which it depends is inoperable. Operational state is disabled.
Degraded	The service available from the resource is degraded in some manner. Operational state is enabled.
Not Installed	The resource is not installed. Operational state is disabled.

Not all the availability status values apply to all resources.

Alarm Status

Alarm status provides further details about alarms issued for a resource and the actions underway to correct them. It is set valued and has the value of the empty set when none of the defined values apply. It has the following defined values:

Under repair	The resource is being repaired.
Critical	One or more critical alarms have been detected in the resource and have not been cleared.
Major	One or more major alarms have been detected in the resource and have not been cleared.
Minor	One or more minor alarms have been detected in the resource and have not been cleared.
Alarm outstanding	One or more alarms have been detected in the resource and have not been cleared. The condition might be disabling.

Relationships of Status Fields

Though availability status places restrictions on the values of operational state, alarm status places no restrictions on the allowable values of operational state. Availability status and alarm status place no restrictions on the values the other can take. This means that there are no combinations of availability status and alarm status that are not valid. Because availability status is the main qualifier of operational state, it is combined with operational state to determine the state of a resource. See *ISO 10164-2* for more information about state management.

Mapping to NetView for AIX States

Table 54 on page 337 shows how the valid combinations of operational state, unknown status, availability status, and alarm status values map to NetView for AIX status. If a column contains **any**, any legal value of the variable is allowed. All combinations not covered in the following table are not valid, and map to an NetView for AIX status of unknown.

Table 54. Mapping Operational State and Status to NetView for AIX Status

Operational State	Unknown Status	Availability Status	Alarm Status	NetView for AIX Status
any	true	any	critical	critical
any	true	any	other than critical	unknown
enabled	false	empty set	any	normal
any	false	off duty	any	normal
disabled	false	not installed	any	unmanaged
disabled	false	off line	any	marginal
disabled	false	dependency	any	critical
enabled	false	degraded	critical	critical
enabled	false	degraded	other than critical	marginal
disabled	false	failed	any	critical
disabled	false	power off	any	critical
enabled	false	in test	any	marginal

Using the nvotStatusType Structure

The status change routines of the GTM API, which are described in "Using the Status Change Routines" on page 353, use the structure **nvotStatusType**. The status values defined in this structure are the same as those used by the NetView for AIX program. The following table shows, for each status defined in **nvotStatusType**, the values that will be assumed by GTM for the status variables.

Table 55. Mapping nvotStatusType to Other Status Values

nvotStatusType	Operational State	Unknown Status	Availability Status	Alarm Status
STATUS_UNKNOWN	DISABLED	TRUE	EMPTY_SET	MINOR
STATUS_NORMAL	ENABLED	FALSE	EMPTY_SET	MINOR
STATUS_CRITICAL	ENABLED	FALSE	DEGRADED	CRITICAL
STATUS_MARGINAL	ENABLED	FALSE	IN_TEST	MINOR
STATUS_UNMANAGED	DISABLED	FALSE	NOT_INSTALLED	MINOR

The STATUS_UNMANAGED status should be used only when creating objects.

Topology Objects in the NetView for AIX Object Database

The GTM creates objects in the NetView for AIX object database based on the topology information it receives. The objects belong to the following categories:

- Simple Connection
- Arc
- Vertex
- Graph

Each of the objects has a name field. Applications may use the name field as a key into the object database.

Other traps or API calls may cause GTM to add fields to existing objects, merge objects together, or split objects apart. The traps that cause these actions are :

- SAP
- Member
- IP Discovery

Simple Connections

Simple connections have one name field, which is "*localEndpointProtocol* Connection". The value of the field is "*localEndpointName simpleConnIndexId*".

OVW Field Name	**Type**
isSimpleConnection	Boolean Cap
is*localEndpointProtocol*	Boolean Cap Loc
localEndpointProtocol Connection	String Name Loc
XXMAP Status	Enumeration of OV Statuses
XXMAP Operational State	Enumeration of Operational States
XXMAP Unknown Status	Boolean
XXMAP Availability Status	Bitmask of Availability Statuses
XXMAP Alarm Status	Bitmask of Alarm Statuses
XXMAP Management Extension	String
XXMAP Management Address	String
XXMAP Protocol	String
XXMAP Index Id	Int32
XXMAP Name Binding	Int32

Arcs

The arc name field is "*aEndpointProtocol zEndpointProtocol* Arc". The value of the field is "*aEndpointName zEndpointName arcIndexId*".

OVW Field Name	**Type**
isArc	Boolean Cap
is*aEndpointProtocol*	Boolean Cap Loc
is*zEndpointProtocol*	Boolean Cap Loc
aEndpointProtocol zEndpointProtocol Arc	String Name Loc
XXMAP Status	Enumeration of OV Statuses
XXMAP Operational State	Enumeration of Operational States
XXMAP Unknown Status	Enumeration of Unknown Statuses
XXMAP Unknown Status	Boolean
XXMAP Availability Status	Bitmask of Availability Statuses

XXMAP Alarm Status	Bitmask of Alarm Statuses
XXMAP Management Extension	String
XXMAP Management Address	String
XXMAP Protocol	String
XXMAP Protocol z	String
XXMAP Index Id	Int32
XXMAP Name Binding	Int32
XXMAP Name a	String
XXMAP Name z	String

Vertices

The vertex name field is "*vertexProtocol* Address". The value of the field is "*vertexName*".

OVW Field Name	**Type**
isVertex	Boolean Cap
XXMAP Location	String
XXMAP Protocol List	String List
XXMAP SAPs Used List	Int32 List (Set by SAP traps)
is*vertexProtocol*	Boolean Cap Loc
vertexProtocol Address	String Name Loc
vertexProtocol Status	Enumeration of OV Statuses
vertexProtocol Operational State	Enumeration of Operational States
vertexProtocol Unknown Status	Boolean
vertexProtocol Availability Status	Bitmask of Availability Statuses
vertexProtocol Alarm Status	Bitmask of Alarm Statuses
vertexProtocol Management Extension	String
vertexProtocol Management Address	String

Graphs

The graph name field is "*graphProtocol* Name". The value of the field is "*graphName*".

OVW Field Name	**Type**
isBox	Boolean Cap (True when graphType = BOX)
isGraph	Boolean Cap (True when graphType = GRAPH)
isRootGraph	Boolean Cap
XXMAP Location	String
XXMAP Protocol	String

XXMAP Protocol List	String List (Exists only when isNode = TRUE)
is*graphProtocol*	Boolean Cap Loc
graphProtocol Name	String Name Loc
XXMAP Layout Algorithm	Enumeration of Graph layout algorithms
graphProtocol Management Extension	String
graphProtocol Management Address	String

Members

Member traps affect Box Graph and Vertex objects.

When a Vertex member component is added to a Box Graph member, the *TopM Node ID* field is added to the Vertex object. The value of this field is the OVwObjectId of the Box Graph member.

Merging Box Graphs

If a Vertex member component is added to a Box Graph member, and that Vertex has a *TopM Node ID* that was previously set to a value that is not the OVwObjectId of the current Box Graph member, then the Vertex is a member of two different Box Graphs. Since the Vertex has the semantics of an interface card, and a Box Graph has the semantics of a computer, this situation implies that one interface card exists in two computers. This situation is impossible; we must merge the two Box Graphs into one Box Graph. We do this by moving all of the fields from the object identified by the Vertex's *TopM Node ID* to the object identified by the Box Graph member, deleting the object identified by the Vertex's *TopM Node ID*, then synchronizing the Box Graph member.

An *OVwMapName protocol* Symbol List field is added to the member component if the member component is either a Vertex or Box Graph. This field contains a list of the symbols that exist for the Vertex or Box Graph object.

OVW Field Name	**Type**
TopM Node ID	Integer (added to a member component Vertex when the member is a Box Graph)
XXMap Protocol Members	Integer List (added to a member component Vertex when the member is a Box Graph)
OVwMapName vertexProtocol Symbol List	Integer List (added to member component Vertices)
OVwMapName graphProtocol Symbol List	Integer List (added to member component Box Graphs)

SAPs

SAP traps affect Vertex and Box Graph objects.

SAP-induced changes to Vertex

Adding SAPs to a Vertex sometimes cause Vertex objects to be merged. Removing SAPs from a Vertex sometimes cause Vertex objects to be split.

Adding SAPs to a Vertex: If the Vertex described by "*sapProtocol sapName*" was previously added with a Vertex add trap, then two objects exist: one with field "*sapVertexProtocol* Address" set to "*sapVertexName*", and one with field "*sapProtocol* Address" set to "*sapName*". In this case, all of the fields on the object with field "*sapProtocol* Address" set to *sapName* are moved to the other Vertex object, and then it is deleted. This operation is called *merging*.

If the SAP Service Type is **providing**, a "*sapVertexProtocol* SAP Protocols Provided" List field is added to the Vertex object. This list contains all of the protocols provided by a given Vertex.

If the SAP Service Type is **using**, the Vertex's "XXMAP SAPs Used List" item that corresponds with the Vertex's "XXMAP Protocol List" item equal to "*sapVertexProtocol*" is set to "*sapProtocol*".

For example, if two vertices existed with the following field values :

```
Vertex 1

XXMAP Protocol List          XXMAP SAPs Used List
-------------------          --------------------
 9                            0

Vertex 2

XXMAP Protocol List          XXMAP SAPs Used List
-------------------          --------------------
56                            0
```

A SAP Add trap with "*sapVertexProtocol*" = 56, which uses "*sapProtocol*" = 9, will result in Vertex 1 being deleted and all fields from Vertex 1 moved to Vertex 2. In addition, the XXMAP SAPs Used List would be updated to show that protocol 56 uses 9, and 9 does not use any other protocol.

```
Vertex 2

XXMAP Protocol List          XXMAP SAPs Used List
-------------------          --------------------
56     (Uses  ----------->)    9
 9     (Uses  ----------->)    0
```

The name of the provided or used SAP is also added to the vertex.

OVW Field Name	Type
sapVertexProtocol SAP Protocols Provided	Int32 List (Added if the SAP is provided.)
XXMAP SAPs Used	Int32 List (Changes only if the SAP is used. This field is added when the Vertex is created.)
sapProtocol Address	String Name Loc

Removing SAPs from a Vertex: In general, removing a SAP causes actions opposite to those caused by adding a SAP. If the Vertex has been merged due to a adding a SAP, as described in "Adding SAPs to a Vertex," then removing that SAP causes the Vertex object to be split into two Vertex objects. The object is split based on information contained in the "XXMAP Protocol List", "XXMAP SAPs Used List", and "*sapVertexProtocol* SAP Protocols Provided List" fields.

SAP-induced changes to Box Graphs

If adding a SAP causes two Vertex objects to be merged, as described in "Adding SAPs to a Vertex" on page 341, and the values of the *TopM Node ID* fields of the two Vertices are different, then the Box Graph objects identified by the *TopM Node ID* fields are merged. The situation that mandates this merge is described in "Merging Box Graphs" on page 340.

IP Discovery

The IP discovery, IP topology and IP map programs correlate IP nodes and IP interfaces with boxes and vertices added by the GTM.

Chapter 21. Communicating with the General Topology Manager

This chapter describes the ways in which your program can interact with the General Topology Manager. It includes the following topics:

The Discovery Process

You can integrate your open-topology application with the NetView for AIX discovery process, or you can locate agents supporting your protocol through your application. This section describes the steps necessary to have the NetView for AIX discovery process help you to identify your agents and start your management application. It concludes with some information about using your own discovery process.

There are several tasks you must complete in order to enable the NetView for AIX program to monitor your non-IP network. Each of the following tasks is explained in this section:

- Assign an object identifier (oid) to be used for all interface nodes that use a given non-IP protocol.
- Create a proxy agent to represent the objects in the non-IP network. This agent must support the oid assigned for this protocol.
- Create a daemon that can interact with the agent, and that will send information to the NetView for AIX General Topology Manager.
- Code an entry in the oid_to_command file, specifying the start command and other information related to your daemon.

Note: This chapter refers to an agent that supports your protocol's oid and a daemon that interacts with that agent and with the gtmd daemon. This description is intended to clarify the different roles that must be fulfilled within this model of communication. If combining these roles into one piece of software meets your needs better, you can take that approach and think of the agent and daemon described here as functions within your code.

Discovering Nodes

When the NetView for AIX network monitor daemon, netmon, discovers an IP node, it issues a trap, which is received by the NetView for AIX non-IP topology discovery daemon, noniptopod. The netmon daemon passes the IP Address of the discovered node to the noniptopod daemon. The noniptopod daemon reads the oid_to_command file. For each entry in this file, the noniptopod daemon issues an SNMP GET request, using the oid from the file entry, to the IP Address of the newly-discovered node. If it receives a non-null response to any of these requests, the noniptopod daemon issues the command that matches the oid for which a value was received.

Note: If your interface node is not in the discovered region of the NetView for AIX management station, add it to your netmon seed file to ensure its discovery. Seed files are described in the *NetView for AIX Administrator's Guide*.

Adding Your Daemon to the oid_to_command File

Add an entry to the oid_to_command file, describing how your daemon is invoked, so that the noniptopod daemon can invoke your daemon whenever an agent that supports that oid is discovered. In this file, specify:

- The oid assigned to your interface node.
- The host name where your daemon resides, if it is not local. Do not enter the local host name if your daemon is local; this will cause an error in invoking the daemon.
- The command to be issued to invoke your daemon.
- The command to be issued to stop your daemon. This entry is not required, but it is useful in ensuring an orderly shutdown when necessary.

The format of the file entry is specified in the man page for the oid_to_command file.

If you add a protocol not supported by the Internet MIB-2, you must also add your protocol information to two other data files, the snmp_fields data file and the oid_to_protocol data file. These files are described next.

The snmp_fields Data File

The /usr/OV/fields/$LANG/snmp_fields file is used by the xxmap application to convert from an integer specifying the vertex protocol in the NetView for AIX topology MIB to a string that represents the vertex protocol name. The file shipped with the NetView for AIX program lists several protocols in an enumeration list. The operator or other applications are free to add to this file for user-defined integers and protocols. The first enumeration, `Unset`, has the value 0, and each succeeding enumeration has a value of one larger than the previous enumeration; `Other` = 1, `Regular 1822` = 2, and so on. The file has the following format:

```
Field "SNMP ifType" {
        Type    Enumeration;
        Flags   locate;
        Enumeration     "Unset",
                        "Other",
                        "Regular 1822",
                        "HDH 1822",
                        "DDN X.25",
                        "RFC 877 X.25",
                        "Ethernet CMSACD",
                        "IEEE 802.3 CMSACD",
}
```

The oid_to_protocol Data File

The /usr/OV/conf/oid_to_protocol file is used by the xxmap application to convert from an oid provided in the NetView for AIX topology MIB to a string that represents the graph protocol name. The file shipped with the NetView for AIX program lists several protocols in an enumeration list. The operator or other applications are free to add to this file for user-defined oids and protocols. The oid_to_protocol file has the following format:

```
# Comment
"SNMP ifType"=1.3.6.1.2.1.2.2.1.3

${SNMP ifType}.1="Other"
${SNMP ifType}.2="Regular 1822"
${SNMP ifType}.3="HDH 1822"
${SNMP ifType}.4="DDN X.25"
${SNMP ifType}.5="RFC 877 X.25"
${SNMP ifType}.6="Ethernet CMSACD"
```

Creating an LRF for Your Agent and Daemon

You should create a local registration file (LRF) for the agent that represents your non-IP network. See “The Local Registration File” on page 19 for information about local registration files. You can also create an LRF for your daemon. Whether you should create an LRF for your daemon depends on when you want your daemon to run:

- If you want your daemon to be started whenever the NetView for AIX program is started, create an LRF for your daemon and use the ovaddobj command to add your daemon to the NetView for AIX startup file. Refer to the **ovaddobj** man page for more information.
- If you want your daemon to be invoked only when an interface node with the appropriate oid is discovered, do not create an LRF for your daemon.

Figure 18 on page 346 illustrates the data flow for the process of discovering an agent that supports an oid and retrieving data from that agent.

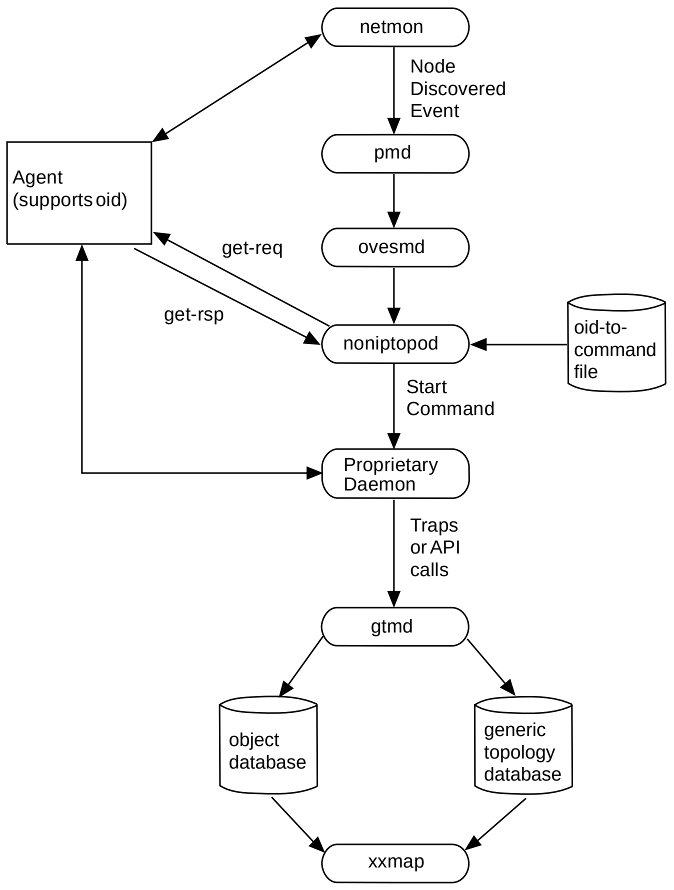

Figure 18. The Discovery Process

Other Discovery Methods

If you prefer not to integrate your management application with the NetView for AIX discovery process, you can discover agents that support your protocol from within your application. For example, if you support only one oid, you can write an application that always runs and have it register for node-discovery traps. When a trap is received, you can issue a get request for your oid to determine whether the agent supports it.

Another possibility, if your network is very stable, is to maintain a list of IP addresses for the nodes managed by your proxy agents. Then your application can interrogate those agents for information.

Note that if you do not use the NetView for AIX discovery process to activate your daemon, you must either add your daemon to the ovsuf file, so that it will be started when the NetView for AIX program is started, or provide some other means, such as a user interface, for starting your application. Even if you perform the discovery process, your application must use the NetView for AIX General Topology Manager to communicate topology information to the NetView for AIX program.

Creating and Updating a Topology

Most of the terminology used by the GTM comes from graph theory. See "Understanding Key Terms" on page 317 for a review of key terms. The sample below shows how to represent a topology using GTM. This example will use only the main components defined in the Open Topology MIB.

Because the NetView for AIX program uses a hierarchical model to display topology information, the first thing we have to create is a root object. This object will be represented as a graph and it will be displayed in the root submap. The application can define the label, icon and some other attributes of this graph. After creating the root graph, all the topology information should be created under this root.

Suppose your application is a LAN discovery. Under the root graph you have another graph that represents a token ring network, so that by opening the root graph submap you can see another graph representing the token ring network.

This network contain stations that will be represented in gtmd by boxes. Although boxes and graphs have the same attributes they are different. We can consider that a box always represents a resource like a computer. Opening the token ring submap you can see stations.

To complete our sample, suppose that each station contains a token-ring interface card. The interface card will be represented by a vertex in gtmd, so opening each station submap you can see one token ring interface card.

The following steps shows how to create the sample described above in the GTM database.

Step 1. Create a root graph ***LAN_Root***

Step 2. Create a graph ***Token_Ring*** to represent the token ring network, and create a relationship between the token ring graph and the root graph: ***LAN_Root*** contains ***Token_Ring***

Step 3. Create a box ***John*** and define the relationship: put box ***John*** inside network ***Token_Ring***

Step 4. Create a box ***Paul*** and define the relationship: put box ***Paul*** inside the network ***Token_Ring***

Step 5. Create a box ***George*** and define the relationship: put box ***George*** inside the network ***Token_Ring***

Step 6. Create a box ***Ringo*** and define the relationship: put box ***Ringo*** inside the network ***Token_Ring***

Step 7. Create a vertex ***tr_1*** and define the relationship: put vertex ***tr_1*** inside the box ***John***

Step 8. Create a vertex ***tr_2*** and define the relationship: put vertex ***tr_2*** inside the box ***Paul***

Step 9. Create a vertex ***tr_3*** and define the relationship: put vertex ***tr_3*** inside the box ***George***

Step 10. Create a vertex ***tr_4*** and define the relationship: put vertex ***tr_4*** inside the box ***Ringo***

The sample in /usr/OV/prg_samples/nvot shows the sequence of routines that creates the topology above using GTM API routines.

Using the same sample, let us now connect the network. Suppose that station ***John*** is connected to station ***Paul***, ***Paul*** is connected to ***George***, ***George*** is connected to ***Ringo*** and ***Ringo*** is connected to ***John***. The connections are represented in GTM by arcs.

The arcs should be inside a graph to be displayed, so the four arcs mentioned above will be inside the graph ***Token_Ring***.

In order to complete the whole sample, let us show one of the correlation mechanisms used by GTM. Suppose that the token ring interface card ***tr_1*** inside the station ***John*** is using the physical address 10005AA825E8 and suppose that IP Topology discovery has discovered an IP node ***Lennon*** that provides this token ring interface card with physical address 10005AA825E8.

If ***John*** has a token ring card using address 10005AA825E8 and ***Lennon*** is providing a token ring card with the same address, then ***John*** and ***Lennon*** are the same physical station.

Using the SAP table of the NetView for AIX Open Topology MIB, you can correlate objects that represent related resources. In the sample above, the token ring card ***tr_1*** and the IP interface card will be displayed together by both ipmap and xxmap.

Topology Update

After the initial discovery process, your agent must detect and report changes to the topology of your network. Your agent should inform your daemon of these changes, and then your daemon must notify the NetView for AIX program.

The traps that are available for this purpose are described in “Open Topology MIB Traps” on page 332 and listed in Table 51 on page 332.

Note: In order to have new traps properly correlated with existing nodes, you must send the vertexProtocol and vertexName or the graphProtocol and graphAddress as the first two variables in the trap. If you use the routines of the GTM API to update your topology, it is crucial to use the correct index variables to ensure that updates are applied to the correct elements.

Sending Information to the General Topology Manager

When your proprietary daemon is invoked, it should obtain topology data from the agent representing the non-IP objects. Communications between your agent and daemon can be in any format. You can choose either of two methods for passing information from your daemon to the NetView for AIX program:

- You can establish an SNMP session with the NetView for AIX program and use the Open Topology MIB and the NetView for AIX SNMP API or SNMP commands to send enterprise-specific SNMP traps containing open topology information.
- You can use the NetView for AIX GTM API to open a socket interface with the gtmd daemon and to pass open topology information to gtmd.

You can use a combination of the two methods if this best suits your purpose. The GTM API now supports all of the tables defined in the Open Topology MIB. “Using the GTM API” on page 350 lists the tables supported by the GTM API.

Both methods send information to the gtmd daemon, which stores the information in a data base where it is available to the xxmap display application. Both methods are described here.

Using the Trap Interface

In order to use the NetView for AIX GTM trap interface, your daemon must establish an SNMP session with the NetView for AIX program. Your daemon then passes information to the gtmd daemon in the form of enterprise-specific SNMP traps defined in the Open Topology MIB. These traps are described in “Open Topology MIB Traps” on page 332.

Trap Interface Example

The following sample demonstrates the use of the trap interface to create a graph with two other graphs as members and one arc between them, which is a member arc. The snmptrap command is used to send the traps.

```
# node = manaca
# enterprise = .1.3.6.1.4.1.2.6.3.1
# agent-addr = 9.179.2.113 (manaca.sumare.ibm.com)

#Create a root graph
#specific-trap = 0x70000015 (1879048213)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048213 1  \
.1.3.6.1.4.1.2.5.3.4.1.1.2 ObjectIdentifier  2.2.1.3.9                       \
.1.3.6.1.4.1.2.5.3.4.1.1.3 OctetString       "Root_Graph"                    \
.1.3.6.1.4.1.2.5.3.4.1.1.1 Integer           3                               \
.1.3.6.1.4.1.2.5.3.4.1.1.4 Integer           3                               \
.1.3.6.1.4.1.2.5.3.4.1.1.7 OctetString       "world.gif"                     \
.1.3.6.1.4.1.2.5.3.4.1.1.6 OctetString       "Root_Graph_Location"           \
.1.3.6.1.4.1.2.5.3.4.1.1.8 ObjectIdentifier  2.2.1.3.9                       \
.1.3.6.1.4.1.2.5.3.4.1.1.9 OctetString       "Root_Graph_Management_Address" \
.1.3.6.1.4.1.2.5.3.4.1.1.10 ObjectIdentifier 2.2.1.3.9.11                    \
.1.3.6.1.4.1.2.5.3.4.1.1.11 Integer          1

#Create a bus graph
#specific-trap = 0x70000015 (1879048213)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048213 1  \
.1.3.6.1.4.1.2.5.3.4.1.1.2 ObjectIdentifier  2.2.1.3.6                       \
.1.3.6.1.4.1.2.5.3.4.1.1.3 OctetString       "Bus_Graph"                     \
.1.3.6.1.4.1.2.5.3.4.1.1.1 Integer           3                               \
.1.3.6.1.4.1.2.5.3.4.1.1.4 Integer           4                               \
.1.3.6.1.4.1.2.5.3.4.1.1.7 OctetString       "newyork.gif"                   \
.1.3.6.1.4.1.2.5.3.4.1.1.6 OctetString       "Bus_Graph_Location"            \
.1.3.6.1.4.1.2.5.3.4.1.1.8 ObjectIdentifier  2.2.1.3.6                       \
.1.3.6.1.4.1.2.5.3.4.1.1.9 OctetString       "Bus_Graph_Management_Address"  \
.1.3.6.1.4.1.2.5.3.4.1.1.10 ObjectIdentifier 2.2.1.3.6.11

#Create bus graph as root graph member
#specific-trap = 0x7000001E (1879048222)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048222 1 \
.1.3.6.1.4.1.2.5.3.4.4.1.2 ObjectIdentifier  2.2.1.3.9                      \
.1.3.6.1.4.1.2.5.3.4.4.1.3 OctetString       "Root_Graph"                  \
.1.3.6.1.4.1.2.5.3.4.4.1.4 Integer           1                             \
.1.3.6.1.4.1.2.5.3.4.4.1.6 ObjectIdentifier  2.2.1.3.6                     \
.1.3.6.1.4.1.2.5.3.4.4.1.7 OctetString       "Bus_Graph"

#Create a 6611 router
#specific-trap = 0x70000015 (1879048213)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048213 1  \
.1.3.6.1.4.1.2.5.3.4.1.1.2 ObjectIdentifier  2.2.1.3.29                     \
.1.3.6.1.4.1.2.5.3.4.1.1.3 OctetString       "6611_Router"                  \
.1.3.6.1.4.1.2.5.3.4.1.1.1 Integer           4                              \
.1.3.6.1.4.1.2.5.3.4.1.1.4 Integer           7                              \
.1.3.6.1.4.1.2.5.3.4.1.1.7 OctetString       "ncarolina.gif"                \
.1.3.6.1.4.1.2.5.3.4.1.1.6 OctetString       "6611_Router_Location"         \
.1.3.6.1.4.1.2.5.3.4.1.1.8 ObjectIdentifier  2.2.1.3.29                     \
.1.3.6.1.4.1.2.5.3.4.1.1.9 OctetString       "6611_Router_Address"          \
.1.3.6.1.4.1.2.5.3.4.1.1.10 ObjectIdentifier 2.2.1.3.29.10

#Create 6611 router as root graph member
#specific-trap = 0x7000001E (1879048222)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048222 1 \
.1.3.6.1.4.1.2.5.3.4.4.1.2 ObjectIdentifier  2.2.1.3.9                     \
.1.3.6.1.4.1.2.5.3.4.4.1.3 OctetString       "Root_Graph"                  \
.1.3.6.1.4.1.2.5.3.4.4.1.4 Integer           2                             \
.1.3.6.1.4.1.2.5.3.4.4.1.6 ObjectIdentifier  2.2.1.3.29                    \
.1.3.6.1.4.1.2.5.3.4.4.1.7 OctetString       "6611_Router"

#Create an arc between bus graph and 6611 router
#specific-trap = 0x7000000E (1879048206)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048206 1 \
.1.3.6.1.4.1.2.5.3.3.1.1.4 ObjectIdentifier  2.2.1.3.6                     \
.1.3.6.1.4.1.2.5.3.3.1.1.5 OctetString       "Bus_Graph"                   \
.1.3.6.1.4.1.2.5.3.3.1.1.7 ObjectIdentifier  2.2.1.3.29                    \
.1.3.6.1.4.1.2.5.3.3.1.1.8 OctetString       "6611_Router"                 \
.1.3.6.1.4.1.2.5.3.3.1.1.9 Integer           1

#Create a member arc
#specific-trap = 0x70000018 (1879048216)
/usr/OV/bin/snmptrap manaca .1.3.6.1.4.1.2.6.3.1 9.179.2.113 6 1879048216 1 \
.1.3.6.1.4.1.2.5.3.4.2.1.2 ObjectIdentifier  2.2.1.3.9                     \
.1.3.6.1.4.1.2.5.3.4.2.1.3 OctetString       "Root_Graph"                  \
.1.3.6.1.4.1.2.5.3.4.2.1.4 Integer           1                             \
.1.3.6.1.4.1.2.5.3.4.2.1.5 ObjectIdentifier  2.2.1.3.6                     \
.1.3.6.1.4.1.2.5.3.4.2.1.6 OctetString       "Bus_Graph"                   \
.1.3.6.1.4.1.2.5.3.4.2.1.7 ObjectIdentifier  2.2.1.3.29                    \
.1.3.6.1.4.1.2.5.3.4.2.1.8 OctetString       "6611_Router"                 \
.1.3.6.1.4.1.2.5.3.4.2.1.9 Integer           1
```

Using the GTM API

The NetView for AIX GTM API is a set of routines that enable an application to interact with the General Topology Manager over a socket interface rather than by using traps. To provide reliability of data transfer, the socket interface in the API operates in blocking mode. This means that if the socket buffer is full, the application will be blocked until the request is totally written into the socket buffer, avoiding the loss of information.

With Version 4 of NetView for AIX, the GTM API includes a group of basic routines that give you complete open access to all tables, variables, and operations defined in the NetView for AIX Generic Topology MIB. Each of these basic routines includes a parameter that serves as a pointer to a structure you define. The structure defines all table variables or attributes of the corresponding object on which the operation is being performed. "Example of Defining a Graph Structure" on page 355 has an example of how you define such a structure. This example is part of a sample program called /usr/OV/prg_samples/nvot/nvotBasicSerialUnderlyingArc.c.

The GTM API now supports all the tables defined in the Open Topology MIB. The API routines operate on the following tables:

- Vertex
- Graph
- Arc
- SAP
- Member
- Member Arc
- Additional Member
- Additional Graph

GTM API Routines

The routines in the GTM API are divided into the following groups:

- Basic routines: use these routines to gain completely open access to all tables, variables, and operations defined in the Generic Topology MIB. See "Using the Basic Routines" on page 352 for more information on these routines.
- Integration routines: use these routines to establish the socket connection between your application and the General Topology Manager, to control the creation of objects in the NetView for AIX object database, and to interpret messages returned from API calls. See "Using the Integration Routines" on page 352 for more information on these routines.
- Create routines: use these routines to create new objects in the open-topology database and to establish their relationships with other objects. See "Using the Create Routines" on page 353 for more information on these routines.
- Delete routines: use these routines to delete existing objects from the open-topology database. See "Using the Delete Routines" on page 353 for more information on these routines.
- Status change routines: use these routines to change the status of a vertex or an arc in the open-topology database. See "Using the Status Change Routines" on page 353 for more information on these routines.
- Variable value change routines: use these routines to change the values of variables in the open-topology database. See "Using the Variable Value Change Routines" on page 353 for more information on these routines.
- Get routines: use these routines to retrieve information from the tables in the open-topology database. See "Using the Get Routines" on page 354 for more information on these routines.
- Free routines: use these routines to free memory allocated by some of the get routines. See "Using the Free Routines" on page 354 for more information on these routines.

The names of most of the GTM API routines clearly indicate the function performed by each routine. Consult the man pages for the details of coding each routine.

Using the Integration Routines

There are five integration routines in the NetView for AIX GTM API. Use these routines to control the establishment of your GTM session and its behavior.

nvotInit Use this routine to open a socket connection to the General Topology Manager If your application runs on a different network node from the NetView for AIX program, specify the hostname of the node on which the NetView for AIX program runs. You can also specify details of the handling of arc objects and parent graphs. See the man page for details.

nvotDone
Use this routine to close the connection between your application and the General Topology Manager. If your application is designed to run continually, you may not need to call this routine.

nvotSetSynchronousCreation
Use this routine to indicate whether your application is interested in the ObjectId of each created object. Each item created in the gtm database is also created in the NetView for AIX object database. If you want the ObjectId of each object that is created in the NetView for AIX object database returned to your application, use this routine to say so. Note that the socket interface will run faster if ObjectIds are not returned. The default behavior is not to return ObjectIds.

nvotGetError
Use this routine to retrieve the error code set upon the execution of the last API routine.

nvotGetErrorMsg
Use this routine to display the message string associated with a return code. You can pass the result of the nvotGetError routine into this routine to have the message displayed. See the man page for an example.

Using the Basic Routines

Programmers with some familiarity with the Generic Topology MIB can use the following routines to gain completely open access to all tables, variables, and operations defined in the Generic Topology MIB.

- nvotArcHandler
- nvotGraphHandler
- nvotVertexHandler
- nvotSapHandler
- nvotSimpleConnectionHandler
- nvotUnderlyingArcHandler
- nvotUnderlyingConnectionHandler
- nvotMemberHandler
- nvotMemberArcHandler
- nvotAttachedArcHandler
- nvotAdditionalMemberHandler
- nvotAdditionalGraphHandler

Using the Create Routines

Use the following routines to create new objects in the open topology database and establish their relationships with other objects:

- nvotCreateVertexInGraph
- nvotCreateVertexInBox
- nvotCreateArcInGraph
- nvotCreateRootGraph
- nvotCreateGraphInGraph
- nvotCreateBoxInGraph
- nvotCreateUsingSap
- nvotCreateProvidingSap
- nvotCreateSerialUnderlyingArc
- nvotCreateParallelUnderlyingArc
- nvotCreateGraph
- nvotCreateBox

Using the Delete Routines

Use the following routines to delete existing objects from the open topology database:

- nvotDeleteVertex
- nvotDeleteArc
- nvotDeleteGraph
- nvotDeleteBox
- nvotDeleteUsingSap
- nvotDeleteProvidingSap
- nvotDeleteVertexFromGraph
- nvotDeleteVertexFromBox
- nvotDeleteArcFromGraph
- nvotDeleteGraphFromGraph
- nvotDeleteBoxFromGraph
- nvotDeleteUnderlyingArc

Using the Status Change Routines

Use the following routines to change the status of a vertex or an arc in the open topology database:

- nvotChangeVertexStatus
- nvotChangeArcStatus

Using the Variable Value Change Routines

Use the following routines to change the values of variables in the tables in the open topology database:

- nvotChangeBoxIcon
- nvotChangeBoxLabel
- nvotChangeGraphIcon
- nvotChangeGraphLabel
- nvotChangeVertexIcon
- nvotChangeVertexLabel
- nvotChangeVertexIconInGraph
- nvotChangeVertexLabelInGraph
- nvotChangeVertexIconInBox

- nvotChangeVertexLabelInBox
- nvotChangeArcIconInGraph
- nvotChangeArcLabelInGraph
- nvotChangeGraphIconInGraph
- nvotChangeGraphLabelInGraph
- nvotChangeBoxIconInGraph
- nvotChangeBoxLabelInGraph
- nvotChangeRootGraphIcon
- nvotChangeRootGraphLabel
- nvotChangeGraphBackground
- nvotChangeBoxBackground
- nvotChangeVertexPositionInGraph
- nvotChangeVertexPositionInBox
- nvotChangeGraphPositionInGraph
- nvotChangeBoxPositionInGraph
- nvotChangeUnderlyingArcIcon
- nvotChangeUnderlyingArcLabel
- nvotSetCenterGraphForGraph
- nvotSetCenterBoxForGraph
- nvotChangeVertexDetails
- nvotChangeGraphDetails
- nvotChangeBoxDetails
- nvotChangeArcDetails

Using the Get Routines

Use the following routines to retrieve information from the tables in the open topology database:

- nvotGetVerticesInGraph
- nvotGetVerticesInBox
- nvotGetArcsInGraph
- nvotGetGraphsInGraph
- nvotGetBoxesInGraph
- nvotGetSapsInVertex
- nvotGetGraphsWhichArcIsMemberOf
- nvotGetGraphsWhichBoxIsMemberOf
- nvotGetGraphsWhichGraphIsMemberOf
- nvotGetGraphsWhichVertexIsMemberOf
- nvotGetBoxesWhichVertexIsMemberOf
- nvotGetArcObjectId
- nvotGetBoxObjectId
- nvotGetGraphObjectId
- nvotGetVertexObjectId

Note that the get routines allocate memory to hold data structures. This memory must be freed by using one of the free routines.

Using the Free Routines

The get routines, which were described in “Using the Get Routines,” allocate memory to hold the data structures that are required to describe open topology objects. Use the free routines to deallocate this memory. There is a free routine for each data structure:

- nvotFreeVertex

- nvotFreeVertexList
- nvotFreeGraph
- nvotFreeGraphList
- nvotFreeBox
- nvotFreeBoxList
- nvotFreeArc
- nvotFreeArcList
- nvotFreeSap
- nvotFreeSapList
- nvotFreeSimpleConnection
- nvotFreeSimpleConnectionList
- nvotFreeUnderlyingConnection
- nvotFreeUnderlyingConnectionList
- nvotFreeUnderlyingArc
- nvotFreeUnderlyingArcList
- nvotFreeMembers
- nvotFreeMembersList
- nvotFreeMemberArcs
- nvotFreeMemberArcsList
- nvotFreeAttachedArcs
- nvotFreeAttachedArcsList
- nvotFreeAdditionalMembers
- nvotFreeAdditionalMembersList
- nvotFreeAdditionalGraph
- nvotFreeAdditionalGraphList

Example of Defining a Graph Structure

The following example shows how you could define a graph structure to be used with the nvotGraphHandler basic routine. This example is taken from the sample program /usr/OV/prg_samples/nvot/nvotBasicSerialUnderlyingArc.c.

```
/********************************************************************/
/* Creating components                                              */
/********************************************************************/

/********************************************************************/
/* Root graph                                                       */
/********************************************************************/

/************************
 ** GRAPH OPERATION - Filling nvotGraph Structure
 ************************/
   nvotGraph    graph;
   operation = CREATE_OPERATION ;
   graph.operation = operation ;
   graph.validAttributes = 0 ;
   graph.graphAttr.graphType = GRAPH ; /* 3 */
   graph.validAttributes |= GRAPHTYPE_ATTR ;

   graph.graphAttr.graphProtocol = strdup( "1.3.6.1.2.1.2.2.1.3.2" ) ;
   graph.validAttributes |= GRAPHPROTOCOL_ATTR ;

   graph.graphAttr.graphName = strdup( "Serial_Underlying_Arc_Basic_Root" ) ;
   graph.validAttributes |= GRAPHNAME_ATTR ;

   graph.graphAttr.layoutAlgorithm = POINT_TO_POINT_LAYOUT ; /* 3 */
   graph.validAttributes |= LAYOUTALGORITHM_ATTR ;

   graph.graphAttr.isRoot = TRUE ;
   graph.validAttributes |= ISROOT_ATTR ;

   rc = nvotGraphHandler (&graph);

   nvotFreeGraph (&graph);
```

Presenting Topology Information to the User

The gtmd daemon stores the topology information that it receives in the NetView for AIX object database and the generic topology database. The xxmap application reads these databases for display and semantic information and presents submaps for non-IP protocols.

When the xxmap application is started, it starts in a synchronization phase. During this synchronization, the NetView for AIX program displays the "synchronizing" message on the status line of the displayed submaps. During synchronization, xxmap brings the displayed maps up-to-date with the topology and object databases.

Editing a Map

The xxmap application enables users to change maps through the NetView for AIX interface. Users can add and delete symbols. However, added symbols will not be verified or stored in the topology database. They will be stored only in the NetView for AIX map database and will exist in the user plane.

Users can delete symbols from the map. The object represented by that symbol will be deleted from the object database only after all symbols representing that object have been deleted. That object which is semantically stored in the topology database will not be deleted. It can be deleted only by the agent that added that particular information.

Symbols deleted by the operator will re-appear the next time xxmap synchronizes that submap. You can use cut-and-paste operations on symbols, but the symbols will exist in the user plane after the paste operation. Care must be taken when cutting and pasting agent-added objects, because these operations are not reflected in the topology database and can cause inconsistencies in the map.

You can write an application that registers for user modifications to the map, and updates the topology database by sending information to the gtmd daemon as if the changes had been detected in the network. Chapter 10, “Map Events and Map Editing” on page 145 describes how to design an application to process user actions.

The Presentation of Protocols

The basis of all protocols, including IP, will be the root submap. On this window, a symbol is displayed for each protocol being managed. Each of these symbols will explode into a submap containing more symbols for that protocol. This submap can also contain symbols that explode into submaps. This aggregation can continue as desired by the user or application.

Each protocol is represented in this tree structure as a separate and distinct branch starting at the root submap. Any application that supports other protocols must build its protocol hierarchy using a graph that has isRoot set to TRUE, graphs, members, and vertices. These can be established with any number of layers. One graph can be a member of several other graphs. However, a vertex can be a member of only one graph of type box, because it represents a physical entity.

Arcs, underlying arcs, simple connections, and underlying simple connections can all be created to represents links between the already defined graphs and vertices. Arcs, graphs, and vertices will appear in NetView for AIX submaps only if they are members of a particular graph.

The NetView for AIX program also provides another submap layout style through the NetView for AIX topology MIB called the pointTopoint layout. This layout allows a directed ring layout with the nodes connected to each other.

Correlation of Protocols

Any protocol running on a particular interface (vertex) or computer (graph) can be correlated with another protocol. Both of these protocols can be discovered by separate agents, with one of the agents knowing it is using both protocols. This information is communicated to the NetView for AIX program using the NetView for AIX topology MIB.

Vertices added by different agents, and later correlated, will become the same object-database object. The graphs that contain these vertices will also become the same object in the object database.

For example, if IP adds a vertex for an IP interface, it is added to the object database and a symbol is drawn on the screen in the node submap. Later, another application discovers the token-ring interface card, which it adds as a vertex. Because the IP vertex knows the token-ring address of the interface card, these two vertices can be correlated. They will be merged to be the same object-database object.

The box objects which contain these vertices will also be merged to become the same object database object. All symbols for the separate objects will be deleted and new symbols will be created for the new merged object.

The result of correlation is that all of the protocols running in the same physical machine will appear in one submap. This means that same computer can appear in several submaps across many protocols but will explode into a single submap containing symbols that represent all of the protocols.

An application or agent must use the SAP table in the NetView for AIX topology MIB to start the correlation process. An application must send a SAP trap that contains the protocol and name of the vertex it has discovered, the protocol and name of the interface to correlate with, and whether this vertex is using a service from the other protocol or providing a service to the other protocol. For example, Protocol IP, Vertex 9.67.7.193 uses the services of Protocol TR, Vertex 10005a65b525. Protocol TR, Vertex 10005a65b525, provides a service to Protocol IP, Vertex 9.67.7.193. Once this information is received, if both vertices are known by the NetView for AIX program, the correlation will take place.

Propagation of Status across Protocols

The xxmap application can be configured to propagate status across all protocols or only within each protocol. The default is to propagate status within each protocol only, so that a failure in any resource will be shown only on the view showing the protocol in which the failure occurred. The user can change the propagation method so that the status of a resource depends on the status of all of its component resources, regardless of protocol. This change can be made permanent by changing the application registration file for xxmap, which is /usr/OV/registration/C/xxmap. The XXMAP Protocol Status field indicates whether status is propagated for each protocol. The default value is True; changing this value to False will cause the xxmap application to propagate status across protocols. The user will still have the option to change the status-propagation method through the graphical user interface.

Switching among Protocols

Another function of the xxmap application is to switch among different protocol branches that contain the same computer or interface. From the View menu item, the Protocols menu item will display a dialog box that lists all of the protocols running on a particular machine, and its interfaces if the symbol represents a machine. On an interface card, the Protocols menu item shows all of the protocols running on that interface. Along with the protocols, it displays the address or name of the object within that protocol and its current status. When a particular protocol is selected in that dialog box, a list of submaps that contain this object within that protocol is listed. From there, a submap can be displayed by clicking on the submap name.

Appendix. Migrating Version 3 Applications

If you have created an application for NetView for AIX Version 3 the migration to Version 4 of the NetView for AIX program should require only minimal changes. The changes you will have to make depend on which application programming interfaces (APIs) and files you have used in creating your application. This chapter is designed to help you understand the changes that will be required to enable your application to run with Version 4 of the NetView for AIX program. For more information on all the enhancements made in Version 4 of the NetView for AIX program, see *NetView for AIX Concepts: A General Information Manual*.

Packaging for a Client/Server Environment

Depending on how extensive your application is and how closely integrated to NetView for AIX, you may need to repackage your application and use the new client/server APIs in your application. Applications that are self-contained and are simply launched from the menu bar may be able to migrate with no changes. Applications that include daemons or an EUI that is closely integrated into NetView for AIX may need to be modified so that some of the function installs on a server, and some of the function is installed on clients. See "Developing Applications for a Client/Server Environment" on page 11 for more information about packaging implications in a client/server environment.

Using NetView for AIX Security

Applications that migrate to Version 4 can do so without making any changes to use NetView for AIX security services. However, if NetView for AIX security is on, your application will not have any security settings and will be unsecured. It is recommended that you ship a minimum security configuration for your application. Chapter 4, "Integrating Your Application with NetView for AIX Security Services" on page 43 explains how to integrate your application with the NetView for AIX security server.

Using NetView for AIX Collections

NetView for AIX now provides a collection facility through which an administrator can perform actions or run applications against a group of objects. The objects are selected based on a policy the administrator defines. There are several nvCol* APIs for integrating an application with the Collection Facility. See the *Programmer's Reference* for more information about these APIs.

Part 4. Glossary, Bibliography, and Index

Glossary

This glossary includes terms and definitions from:

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The ANSI/EIA Standard—440-A, *Fiber Optic Terminology*. Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The Network Working Group Request for Comments: 1208.
- The *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *Object-Oriented Interface Design: IBM Common User Access Guidelines*, Carmel, Indiana: Que, 1992.

The following cross-references are used in this glossary:

Contrast with: This refers to a term that has an opposed or substantively different meaning.

Synonym for: This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the glossary.

Synonymous with: This is a backward reference from a defined term to all other terms that have the same meaning.

See: This refers the reader to multiple-word terms that have the same last word.

See also: This refers the reader to terms that have a related, but not synonymous, meaning.

Deprecated term for: This indicates that the term should not be used. It refers to a preferred term, which is defined in its proper place in the glossary.

A

action. (1) In the AIX Operating System, a defined task that an application performs. An action modifies the properties of an object or manipulates the object in some way. (2) An operation on a managed object, the semantics of which are defined as part of the managed object class definition.

active. (1) The state a resource is in when it has been activated and is operational. Contrast with *inoperative*. (2) In the AIX Operating System, pertaining to the window pane in which the text cursor is currently positioned.

adapter. (1) A part that electrically or physically connects a device to a computer or to another device. (2) Hardware card that allows a device, such as a PC, to communicate with another device, such as a monitor, a printer, or other I/O device.

address. See *internet address*.

agent. In the TCP/IP environment, a process running on a network node that responds to requests and sends information.

aggregate. In programming languages, a structured collection of data objects that form a data type. (I)

AIX. Advanced Interactive Executive.

AIX Operating System. (1) IBM's implementation of the UNIX Operating System. The AIX Operating System runs on the RISC System/6000 system. (2) See *UNIX Operating System.*

AIX NetView/6000. An abbreviated name for AIX SystemView NetView/6000.

AIX NetView Service Point. (1) A licensed program that operates in the AIX and UNIX environments. The AIX NetView Service Point functions as a gateway in an unattended environment. (2) A workstation-based IBM licensed program through which application programs can be used to monitor, manage, and diagnose problems in non-SNA networks and communication devices.

AIX SystemView NetView/6000. A comprehensive management tool for heterogeneous devices on Transmission Control Protocol/Internet Protocol (TCP/IP) networks. The NetView for AIX program can use the IBM AIX Service Point program to communicate with the NetView and NETCENTER programs.

alert. (1) An error message sent to the system services control point (SSCP) at the host system. (R) (2) In the NetView for AIX program, selected traps are converted to alerts that are then forwarded to the NetView or NETCENTER programs for handling. (3) In the NetView and NETCENTER programs, a high-priority event that warrants immediate attention.

API. Application programming interface.

application plane. (1) Submaps contain two layers, or planes, on which symbols are displayed. The application plane displays symbols of objects that are managed by at least one network or system management application. Symbols of an object on the application plane appear without shading, directly against the submap background. (2) See *user plane* and *background plane*.

application programming interface. (1) A library of routines, provided with a product, which allows customer or vendor programmers to integrate applications with the product. (2) A routine within such a library.

application registration file. A file created by a programmer to integrate an application into the NetView for AIX program by defining its place in the program's menu structure, where help information is found, the number and types of parameters allowed, the command line used to start the application, and other characteristics of a user-written application.

arc. In multiple topology, an arc represents connectivity between vertices or graphs. The connection is independent of either end point.

ASCII. American Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8-bit including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

attribute. (1) Objects are defined by attributes. Object attributes appear as fields in the Object Description dialog box and Attributes dialog boxes for applications. The NetView for AIX windows tool provides a dialog box that displays general attributes and capabilities of objects based on their attributes. (2) See also *managed object* and *managed object class*.

attribute list. A list that displays the attributes that can be set for specific objects. These are global object attributes that are valid for an object across maps. The attributes list box is available in the Add Object, Add Connection, and Describe Object dialog boxes. When adding or describing an object, the attributes associated with the object can be viewed or modified.

automatic layout. Automatic layout enables or disables the layout algorithm. Each submap may have a layout algorithm that determines how symbols are placed on the submap. The following layout algorithms are available:

- Point-to-point
- Star
- Bus
- Ring
- Row/Column
- Tree
- No Layout

B

background plane. The lowest layer or plane in a submap. The background plane provides the background that symbols are viewed against. A background graphic can be placed in the background plane to provide contextual information for viewing symbols.

background process. (1) In the AIX Operating System, a mode of program execution in which the shell does not wait for program completion before prompting the user for another command. (2) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. (3) Contrast with *foreground process*. (4) See also *daemon*.

behavior. (1) A characteristic of a symbol that determines what happens when the symbol is selected with the mouse. Symbol behavior may be either explodable or executable.

Explodable	Double-clicking on the symbol causes a question dialog box to be displayed: The question dialog box provides three choices: use the default submap, modify the submap settings, or cancel this submap creation.
Executable	The symbol executes an application that performs an action on a set of target objects.

(2) See also *executable symbol* and *explodable symbol*.

Berkeley Internet Name Domain (BIND). The Berkeley implementation of the Domain Name System (DNS).

BIND. Berkeley Internet Name Domain.

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (R) (2) An adjustable memory storage space, temporarily reserved for performing input or output, into which data is read or from which data is written. (R)

bus layout. (1) A layout algorithm that displays a bus configuration and shows symbols arranged along the bus. A bus topology shows symbols in a linear layout, such as computers on a bus network segment. (2) See *point-to-point layout, ring layout, tree layout, row/column layout,* and *star layout*. See also *layout algorithm*.

bus segment. (1) A part of a network that represents nodes attached to a single, linear cable that transmits data. (2) An expansion of a selected bus segment from the Network. which shows the hosts, gateways, and devices attached to a LAN cable or segment.

button. A word or picture on the screen that can be selected. Once selected and activated, a button begins an action in the same manner that pressing a key on the keyboard can begin an action. (R)

C

callback routine. A routine specified by an application to be started when a specific event is detected.

callback registration. Identifying or registering a callback routine.

cancel button. (1) Exits the dialog box and resets the values of its contents. (2) See *button*.

card. (1) A unique place to display information that relates to an event. A card provides a repository for information and a fast path to the MIB browser application and the topology map representation of managed objects. Cards are placed in workspaces and can be sent to other users, searched, ordered, and reports can be generated from them. (2) See also *MIB* and *workspace*.

CCITT. (1) Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee. (2) See *Consultative Committee on International Telegraph and Telephone (CCITT)*.

child submap. (1) A submap that represents a detailed view of an object, or the “contents” of an object (called the parent object) on a map. Double-clicking on an explodable symbol that represents the parent object opens the child submap. (2) See also *parent object*.

class. In the AIX Operating System, pertaining to the I/O characteristics of a device. System devices are classified as block or character devices.

click. To press and release a mouse button.

client. (1) In an AIX distributed file system environment, a system that is dependent on a server to provide it with programs or access to programs. (2) See also *server*.

CMIP. Common Management Information Protocol.

CMIS. Common Management Information Services.

CMOT. Common Management Information Protocol over TCP/IP.

command. (1) A request from a terminal for the performance of an operation or the execution of a particular program. (2) A request to perform an operation or run a program. When parameters, values, flags, or other operands are associated with a command, the resulting character string is a single command. (R)

Common Management Information Protocol (CMIP). The protocol elements used to provide the operational and notification services defined by Common Management Information Services. CMIP is part of the Organization for Standardization (ISO) and Open Systems Interconnection (OSI) specification.

Common Management Information Services (CMIS). A suite of operational and notification services used for the management of systems. CMIS is a part of the International Organization for Standardization (ISO) and Open Systems Interconnection (OSI) specification.

Common Management Information Protocols over TCP/IP (CMOT). A protocol interface defined by the Internet Engineering Task Force (IETF) that enables the use of CMIP over a TCP/IP protocol stack.

communications infrastructure. A framework of communication that consists of a postmaster, object registration service, startup file, communication protocols, and application programming interfaces. The communication infrastructure also provides access to event management services and data management services.

community name. A password that must be used for certain SNMP requests.

component. Hardware or software that is part of a functional unit.

compound status. (1) The compound status scheme determines how status is propagated from symbols in child submaps to symbols of the parent object. The combined status of symbols determines the resulting compound status. Compound status can propagate up through multiple levels of submaps in the network map. The compound status setting applies to the entire map. In effect, the status of specific nodes propagates up to a symbol on a higher-level submap. Compound status is configured by using one of three schemes:

- Default
- Propagate Most Critical
- Propagate at Threshold Value

(2) See *default compound status*.

configuration. (1) The manner in which the hardware and software of an information processing system are organized and interconnected. (T) (2) The devices and programs that make up a system, subsystem, or network. (3) The act of organizing and interconnecting the components of an information processing system.

connection. (1) In system communications, a line over which data can be passed between two systems or between a system and a device. (2) A physical or logical link between objects that appears as a line between them on the topology map. For example, the connection line between a gateway and network represents an interface on that network. Multiple connections appear as one line. (3) Synonym for *physical connection*.

connection symbol. (1) A special symbol type that connects two icon symbols (or an icon symbol and a backbone) on a submap. The NetView for AIX program provides the following registered connection symbols:

- Solid line
- Dashed line
- Dotted line

(2) See also *icon*.

Consultative Committee on International Telegraph and Telephone (CCITT). A United Nations Specialized Standards group whose membership includes common carriers concerned with devising and proposing recommendations for international telecommunications representing alphabets, graphics, control information, and other fundamental information interchange issues. (R)

context menu. (1) A menu (also known as a pop-up menu) that provides no visual cue to its presence, but pops-up when operators perform a menu selection with button 3 of a three-button mouse.

control desk. (1) A component of the graphical interface that enables the network operator to group applications instances together. The operator can create multiple control desks through configuration of the X-Defaults file or by using the mouse to drag the control-desk icon from the Tool Palette. (2) See also *tool palette*.

copy. (1) In the NetView for AIX program, a menu item function that copies selected symbols and objects to the cut buffer. To complete the copy operation, select the Paste menu item. (2) See *cut, cut buffer,* and *paste*.

critical status. (1) In the NetView for AIX program, the status state, displayed by a symbol, that indicates a problem with the object. If the status is compound status, it reflects a critical condition in the parent object's child submap. If the status is direct status, it may reflect a critical condition for the symbol or the object. The default color for critical status is red. (2) See *normal status, marginal status,* and *compound status*.

cut. (1) A function used to cut (delete) objects and place them in the cut buffer. The Cut Button can be used in conjunction with the Paste menu item to move objects by pasting them from the cut buffer to a submap (cut-and-paste). (2) See *copy, cut buffer,* and *paste*.

cut buffer. (1) A memory area where symbols and objects that are cut or copied are temporarily stored. The cut buffer enables cut and paste, or copy and paste, operations. (2) See *copy, cut,* and *paste*.

D

daemon. (1) A background process usually started at system initialization that runs continuously and performs a function required by other processes. (2) In the AIX Operating System, a program that runs unattended to perform a standard service. Some daemons are triggered automatically to perform their task; others operate periodically. (3) See also *background process*.

data. A representation of facts, concepts, or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data includes constants, variables, arrays, and character strings.

default. An initial configuration setting. Defaults are supplied when the NetView for AIX program is first run to reduce the amount of time required to start actively managing systems on a network. Users and applications can alter many default settings.

default compound status. When a new map is created, compound status is set to a default value. The default value for compound status causes the graphical interface to propagate status.

delete. (1) An Edit menu function that deletes symbols and objects. A confirmation box is displayed before the deletion is performed. Some objects may be rediscovered and their symbols can be hidden. The Delete function is available for maps, submaps, and snapshots. (2) See also *hide symbol* and *edit menu*.

destination. Any point or location, such as a node, station, or a particular terminal, to which information is to be sent.

device. A mechanical, electrical, or electronic contrivance with a specific purpose.

dialog box. (1) A dialog box provides data fields and buttons for setting controls, selecting from lists, choosing from mutually exclusive options, entering data, and presenting the user with messages. The NetView for AIX dialog boxes are defined by OSF/Motif. (2) A pop-up window that is used primarily to gather user input.

discovery. The automatic detection of network topology changes (for example, new and deleted nodes, new and deleted interfaces).

discriminator. (1) An object that supports network management to enable a system to select operations and event reports relating to other managed objects. (2) See also *filter*.

display. (1) A visual presentation of data. (I) (A) (2) To present data visually. (I) (A) (3) A device or medium on which information is presented, such as a terminal screen. (4) Deprecated term for *panel*.

DOC. Documentation.

domain. (1) That part of a network in which the data processing resources are under common control. (T) (2) In a database, all the possible values of an attribute or a data element. (3) In TCP/IP, the naming system used in hierarchical networks. The domain naming system uses the DOMAIN protocol and the named daemon. (4) In a domain system, groups of hosts are administered separately within a tree-structured hierarchy of domains and subdomains.

domain name. (1) A naming scheme consisting of a sequence of subnames separated by a period. Each section of the domain name is called a label. (2) In TCP/IP, a name of a host system in a network. A domain name consists of a sequence of subnames separated by a delimiter character.

double-click. To press and release a mouse button twice in rapid succession.

drag. To press and hold a mouse button while moving the mouse to move the pointer on the computer display. Symbols can be dragged from the Add Object Palette or Add Connection Palette to a submap to place a symbol on a submap. Dragging is used to select menus, move, and resize submap windows or dialog boxes.

dump. (1) To record, at a particular instant, the contents of all or part of one storage device in another storage device. Dumping is usually for the purpose of debugging. (T) (2) Data that has been dumped. (T) (3) To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer.

dynamic. (1) In programming languages, pertaining to properties that can be established only during the execution of a program; for example, the length of a variable-length data object is dynamic. (I) (2) Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. (3) In NetView for AIX, the contents of windows in the event display function are either dynamic or static. In the dynamic display (workspace), events continue to be added to the cards/list. (4) Contrast with *static*.

E

edit menu. An action bar menu that contains items that enable the user to edit symbols and objects in an open map or submap. Editing includes tasks, such as adding, deleting, and copying.

EMS. Event management services

end user. A person, device, program, or computer system that utilizes a computer network for the purpose of data processing and information exchange. (T)

end-user interface (EUI). See *graphical user interface* and *EUI*.

Enhanced X-Windows Toolkit. (1) In the AIX Operating System, a collection of basic functions for developing a variety of application environments. Toolkit functions manage Toolkit initialization, widgets, memory, events, geometry, input focus, selections, resources, translation of events, graphics contexts, pixmaps, and errors. (2) See also *X-Window System*.

enterprise. An entire business organization. An enterprise may consist of one or more establishments, divisions, plants, warehouses, and so on that require an information system.

enterprise-specific MIB. (1) An SNMP management Information Base (MIB) developed by individual vendors for specific products. Vendors register their private MIBs under the enterprise object identifier subtree. (2) See *MIB*.

entity. (1) In the NetView for AIX program, an element on a network that has semantic attributes. (2) See also *object*.

enterprise-specific trap. (1) An enterprise-defined SNMP trap indicated by generic trap number 6 and a unique specific trap number that denotes an enterprise-unique event.

error. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. (I) (A)

EUI. (1) End-user interface. (2) Synonymous with *graphical interface*.

event. (1) An occurrence of significance to a task, such as an SNMP trap or a NetView for AIX internal event. (2) In the NetView for AIX program, an unsolicited notification from the managed object or SNMP agent that at least one of the following has occurred:

- A threshold limit was exceeded.
- The network topology changed.
- An informational message or an error occurred.
- An object's status changed.
- A node's configuration changed.

(3) In the NetView and NETCENTER programs, a record indicating irregularities of operation in physical elements of a network. (4) A CMIP event report.

event card. In the NetView for AIX program, a graphical representation, resembling a punched card, of the information contained in an event sent by an agent to a manager reflecting a change in the status of one of the agent's managed nodes.

event forwarding discriminator. (1) A discriminator that acts on potential event reports. (2) See *discriminator*.

event management services (EMS). A centralized method of generating, receiving, routing, and logging network events.

EXEC. (1) In the AIX Operating System, to overlay the current process with another executable program. (2) See also *fork*.

exclusive submap. (1) A submap that is created by an application with exclusive rights of control. If an application creates a submap as an exclusive submap, only that application can determine what happens in the application plane of the submap. (2) See *shared submap*.

executable symbol. (1) A symbol configured such that double-clicking on it causes an application to perform an action on a set of target objects. When you change the behavior of a symbol to executable, you choose from a list of registered applications and actions, and you choose a set of objects (target objects) that the application acts upon. You can modify these settings at any time. Executable symbols are useful for easily performing complex network management tasks as often as needed. (2) See *explodable symbol* and *behavior*.

explodable symbol. (1) A symbol configured such that double-clicking on it displays the child submap of the parent object that the symbol represents. The child submap displays the contents of the parent object. If the object the symbol represents has no child submap, a question dialog box appears enabling you to create and configure a child submap. After the submap is created, double-clicking on the symbol opens the child submap. (2) See *executable symbol* and *behavior*.

F

feature. A part of an IBM product that may be ordered separately by the customer. A feature is designated as either special or specify and may be designated also as diskette-only.

field. Fields are the building blocks of which objects are composed. A field is characterized by a field name, a data type (integer, Boolean, character string, or enumerated value), and a set of flags that describes how the field is treated by NetView for AIX. A field can contain data only when it is associated with an object.

filter. (1) In the NetView for AIX program, a set of criteria that determines which events are received by registered applications, selected for displaying, or forwarded to the NetView and NETCENTER programs as alerts. (2) In the NetView program, a function that limits the data that is to be recorded on the database and displayed at the terminal. See *recording filter* and *viewing filter*. (3) In the AIX Operating System, a command that reads standard input data, modifies the data, and sends it to the display screen. (4) A device or program that separates data, signals, or material in accordance with specified criteria. (A)

filtering. In the NetView for AIX program, a process that applies tests to previously identified objects to extract a subset.

foreground process. (1) In the AIX Operating System, a process that must run to completion before another command is issued to the shell. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. (2) In the NetView for AIX program, the xnm application and the applications running under it. (3) Contrast with *background process*.

fork. (1) In the AIX Operating System, to create and start a child process. (2) See also *EXEC*.

function index. (1) An index that enables you to get online help that describes the functions of the graphical interface. You can display the Function index from the Help menu. (2) See also *help menu*.

G

gateway. (1) In the AIX Operating System, an entity that operates above the link layer and translates, when required, the interface and protocol used by one network into those used by another distinct network. (2) A functional unit that interconnects two computer networks with different network architectures. A gateway connects networks or systems of different architectures. A bridge interconnects networks or systems with the same or similar architectures. (T)

(3) In TCP/IP, a device used to connect two systems that use either the same or different communication protocols.

GIF. Graphical Interchange Format.

graph. In multiple topology, a graph represents a collection of vertices, arcs, and other graphs. An IP segment is a graph that contains all IP machines and their connections.

graph-attached arc. In multiple topology, a graph-attached arc represents an arc that connects two graphs from within each of the two graphs.

Graphic Interchange Format (GIF). In the NetView for AIX program, the format used for the background pictures of a network topology map.

graphical user interface. (1) In the NetView for AIX program, the integrating interface application that provides the means for displaying submaps and for integrating network applications. The graphical interface is a single, consistent interface that enables multiple applications to interact. (2) See also *EUI*.

gtmd daemon. A background process that receives generic topology information for the multiprotocol topology functions of the NetView for AIX program.

H

help button. Displays the help entries.

help menu. An action bar menu to provides detailed help information about the NetView for AIX graphical interface. It also provides information about registered applications that are integrated with the graphical interface.

help panel. Information displayed by a system in response to a help request from a user. See *task panel*.

hide symbol. (1) An operation that enables users to prevent symbols from being displayed on a submap. The symbols still exist but are not visible.

highlighting. (1) In the NetView for AIX program, a visual cue showing the nodes or connections that are the output of certain operations. (2) Emphasizing a display element or segment by modifying its visual attributes. (I) (A)

home submap. A starting point on a map. The home submap is the first submap that appears when a map is opened. Each map may have a home submap. When new maps are created, the home submap is the root submap.

host. (1) The primary or controlling computer in the communications network. (R) (2) A computer attached to a network. (R) (3) In TCP/IP, any system that has at least one Internet address associated with it. A host with multiple network interfaces may have multiple Internet addresses associated with it. (4) See also *host processor*.

host processor. (1) A processor that controls all or part of a user application network. (T) (2) In a network, the processing unit in which the access method for the network resides.

I

icon. A graphic symbol, displayed on a screen, that a user can point to with a device, such as a mouse, in order to select a particular function or software application. (T)

ID. Identification.

inoperative. The condition of a resource that has been active, but is not. The resource may have failed, received an INOP request, or been suspended while a reactivate command is being processed.

instance. A concrete realization of an abstract object class. An instance of a widget or gadget is a specific data structure that contains detailed appearance and behavioral information that is used to generate a specific graphical object on-screen at run time. (R)

input/output (I/O). The means by which the subagent determines the value of a MIB variable. For example, there is code to instrument a MIB variable to load the central processing unit (CPU).

input/output (I/O). (1) Pertaining to a device whose parts can perform an input process and an output process at the same time. (I) (2) Pertaining to a functional unit or channel involved in an input process, output process, or both, concurrently or not, and to the data involved in such a process.

interface. A shared boundary between two functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. (T)

Internet. A wide-area network connecting thousands of disparate networks in industry, education, government, and research. The Internet network uses TCP/IP as the standard protocol for transmitting information.

internet address. The numbering system used in TCP/IP Internetwork communications to specify a particular network, or a particular host on that network, with which to communicate. Internet addresses are denoted in dotted decimal form.

internet-level submap. The highest level of the topology map that shows how internet protocol networks or subnets are connected by gateways.

I/O. Input/output

IP. Internet Protocol.

ISO. International Organization for Standardization

K

keyword. (1) In programming languages, a lexical unit that, in certain contexts, characterizes some language construct; for example, in some contexts, IF characterizes an if-statement. A keyword normally has the form of an identifier. (I) (2) One of the predefined words of an artificial language. (3) A name or symbol that identifies a parameter. (4) Part of a command operand that consists of a specific character string, such as DSNAME=.

L

label. A label is used to distinguish a symbol from other symbols on a submap and map. The label is displayed below a symbol. Labels can be assigned or modified at any time by using the Symbol Description dialog box.

LAN. Local area network.

layout algorithm. A method for arranging symbols on a map or submap.

legend. An explanatory list of the symbols, icons, and other components of a network.

link. (1) In data communications, a transmission medium and data link control component that together transmit data between adjacent nodes. (R) (2) In TCP/IP, a communications line. A TCP/IP link may share the use of a communications line with SNA.

local. Pertaining to a device, file, or system that is accessed directly from your system, without the use of a communications line. Contrast with *remote.* (R)

local registration file (LRF). A file that provides information about an agent or daemon, such as the name, the location of the executable code, and details about the objects that an agent manages.

LRF. Local registration file.

M

managed object. (1) An object that is being actively managed. Applications can monitor and manage objects for topology, status, and configuration changes. When you choose to manage an object, the objects in child submaps of the managed object also become managed. An object can be toggled between managed and unmanaged. You can choose which objects to manage, thereby, customizing the management region for any map. (2) See *unmanaged object* and *management region*.

managed object class. A named set of managed objects sharing the same (named) sets of attributes, notifications, management operations (packages), and which share the same conditions for presence of those packages. (I)

management information base (MIB). (1) A set of variable bindings that reflect the current state of an SNMP agent. Extensions to the MIB can be added by a business enterprise. (2) See also *enterprise-specific MIB*.

management region. The set of managed objects on a particular map that defines the extent of the network that is being actively managed. The management region may vary across maps.

manager. The part of a distributed management application that issues requests and receives notifications; that is, uses the services of one or more agents.

map. A set of related submaps that provides a graphical and hierarchical presentation of a network and its systems.

member. In multiple topology, the member indicates that other graphs or vertices are contained within a particular graph.

member arc. In multiple topology, a member arc indicates the arc that is contained within a particular graph.

menu. A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated. (T)

menu bar. A rectangular area at the top of the client area of a window that contains the titles of the standard pull-down menus for that application. (R)

menu item. One of a list of options contained in a menu.

message. (1) An assembly of characters and sometimes control codes that is transferred as an entity from an originator to one or more recipients. A message consists of two parts: envelope and context. (2) Information from the system that informs the user of a condition that may affect further processing of a current program. (R)

MIB. Management Information Base.

mnemonic. A single character of a menu item, often the first letter, that represents a function. The mnemonic is the underlined character.

module. A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to or output from an assembler, compiler, linkage editor, or executive routine.

monitor. (1) A device that observes and records selected activities within a data processing system for analysis. Possible uses are to indicate significant departure from the norm, or to determine levels of utilization of particular functional units. (T) (2) Software or hardware that observes, supervises, controls, or verifies operations of a system. (A)

monitor menu. An action bar menu that provides access to application tools that present information about selected objects. The information consists of node name, description, status, and objects.

motif. See *OSF/Motif*.

mouse. A device that a user moves on a flat surface to position a pointer on a panel. It allows a user to select a choice or function to be performed or to perform operations on a panel, such as dragging or drawing lines from one position to another.

N

navigation tree. A component of the graphical user interface that forms a hierarchy of open submaps illustrating the parent-child relationship. The navigation tree enables the network operator to determine which submaps are currently open and to close, restore, or raise the windows that contain submaps.

NETBIOS. Network Basic Input/Output System

netmon daemon. A background process that discovers and monitors nodes on the network. See also *daemon*.

NetView program. An IBM licensed program used to monitor a network, manage it, and diagnose network problems.

network. (1) An arrangement of nodes and interconnecting branches. (T) (2) A configuration of data processing devices and software connected for information interchange.

network address. (1) An identifier for a node, station, or unit of equipment in a network. (2) An address, consisting of subarea and element fields, that identifies a link, a link station, or a network addressable unit. Subarea nodes use network addresses; peripheral nodes use local addresses. The boundary function in the subarea node to which a peripheral node is attached transforms local addresses to network addresses and vice versa. (3) See also *network name*.

Network Basic Input/Output System (NETBIOS). The standard interface to networks, IBM personal computers, and compatible personal computers, that is used on LANs to provide message, print-server, and file-server functions. The IBM NETBIOS application program interface (API) provides a programming interface to the LAN so that an application program can have LAN communication without knowledge and responsibility of the data link control (DLC) interface.

network name. (1) The symbolic identifier by which end users refer to a network addressable unit, a link, or a link station within a given network. In APPN networks, network names are also used for routing purposes. (2) See also *network address*.

network operator. (1) A person who controls the operation of all or part of a network. (2) In a multiple-domain network, a person or program responsible for controlling all domains.

NFS. Network File System. (R)

node. (1) An end point of a link, or a junction common to two or more links in a network. Nodes can be processors, controllers, or workstations, and they can vary in routing and other functional capabilities. (R) (2) The portion of a hardware component, along with its associated software components, that implements the functions of the seven architectural layers (SNA). (3) In a tree structure, a point at which subordinate items of data originate. (R)

node name. In the NetView for AIX program, the symbolic name assigned to a specific node during network definition.

node-level submap. Contains the addressable resources of a network, such as a gateway, router, workstation, and personal computer.

noniptopod daemon. A background process that is responsible for instigating multiprotocol discovery applications during NetView for AIX operation.

normal status. (1) Indicates that a network object is functioning normally. The default icon symbol color for normal status is green. The default connection symbol color for normal status is black. (2) See *critical status* and *marginal status*.

notification. Information emitted by a managed object relating to an event that has occurred within the managed object. (I)

O

object. (1) In the NetView for AIX program, a generic term for any entity that NetView for AIX discovers and displays on the topology map, or any entity that you add to the topology map. (2) In the AIX object data manager, an instance or member of an object class, conceptually similar to a structure that is a member or array of structures. (3) In the NetView for AIX program, objects convey to the symbol various semantic attributes that represent an entity. (4) See *managed object*. (5) See also *entity* and *symbol*.

object attribute. (1) An object is defined by its attributes. The attributes are fields in which values are stored. For any object, an Object Description dialog box enables you to highlight applications and display attribute values that specific applications use to manage the object. For example, each object has a selection name attribute. The value of any selection name is the textual name of the object. Objects that support SNMP have an IP address attribute. The attribute value of any object supporting SNMP is the actual address. Applications and users may assign new values to object attributes, and may provide new attributes for certain objects. (2) See *managed attribute*. (3) See also *attribute, object,* and *selection name*.

object class. (1) In AIX SystemView NetView/6000, objects are divided into four classes: computer, connector, device, software, location, and cards. (2) In the AIX object data manager, a stored collection of objects with the same definition. Conceptually similar to an array of structures. (3) See also *object class, symbol class*, and *managed object class*.

object ID. The unique name identification of a management information base object.

object instance. (1) A specific object in a particular class of objects. A physical network entity or resource, such as a computer, modem, gateway, bridge, X.25 switch; or an abstract network entity such as an application, LAN manager, subnet manager, agent or proxy. Specific object managers own each object instance in the network. (2) See also *object class.*

object registration service (ORS). (1) In the AIX SystemView NetView/6000 program, a component that creates and maintains a global directory of object managers, their locations, and their protocols. The postmaster daemon uses this directory to route messages and provide location transparency for managers and agents, which eliminates having to hard-code object addresses. (2) See also *pmd daemon*.

OK button. Saves and cancels the value of the dialog box.

online. (1) Pertaining to the operation of a functional unit when it is under the direct control of the computer. (T) (2) Pertaining to a user's ability to interact with a computer.

open systems interconnection (OSI). The interconnection of open systems in accordance with standards of the International Organization for Standardization (ISO) for the exchange of information. (T) (A)

Operating System/2 (OS/2). A set of programs that control the operation of high-speed large-memory IBM personal computers (such as the IBM Personal System/2 computer, Models 50 and above), providing multitasking and the ability to address up to 16MB of memory.

operator. (1) A person who operates a device. (2) A person or program responsible for managing activities controlled by a given piece of software such as MVS, the NetView program, or IMS. (3) A person who keeps a system running. (4) See *network operator*.

ORS. Object registration service.

orsd daemon. (1) A background process that maintains the consistency and integrity for the object registration service. (2) See also *object registration service (ORS)*.

OS/2. Operating System/2

OSF. Open Software Foundation.

OSF/Motif. (1) A graphical interface that contains a tool kit, presentation description language, window manager, and style guideline. (2) See also *Open Software Foundation*.

OSI. Open systems interconnection.

ovesmd daemon. A process of event management services that runs on the management station and is responsible for routing and delivering events to manager applications.

ovspmd daemon. A background process that coordinates the start and stop of the other AIX SystemView NetView/6000 daemons.

P

page. The information displayed at the same time on the screen of a display device.

panel. (1) A formatted display of information that appears on a terminal screen. (2) In computer graphics, a display image that defines the locations and characteristics of display fields on a display surface. (3) See *help panel*. Contrast with *screen*.

parent object. The relationship that an object has with its child submap. Symbols of a parent object can appear on multiple submaps.

parent submap. (1) The view from which an object was expanded. Each segment has a parent Network submap. Each network has the Internet submap for its parent. (2) See also *parent window*.

parent window. (1) In AIX Enhanced X Windows, the window that controls the size and location of its children. If a window has children, it is a parent window. (2) See also *parent submap*.

paste. (1) Used in conjunction with the Cut and Copy menu item to complete a cut-and-paste operation or a copy operation. It retrieves items from the cut buffer and places symbols of objects on a submap of your choice. (2) See also *copy and cut buffer.*

physical connection. (1) A connection that establishes an electrical circuit. (2) In the AIX SystemView NetView/6000 program, a point-to-point connection or multipoint connection. Synonymous with *connection*.

ping. Packet internet groper. Used to test Internet Protocol-level connectivity to a destination by sending an ICMP echo request and waiting for a response.

pmd daemon. (1) A background process that centralizes the external communications for all applications and processes. The pmd daemon contains SNMP and CMOT protocol stacks to enable SNMP and CMOT to receive and send information. (2) See also *postmaster*.

point-to-point layout. (1) A layout that shows an arbitrarily interconnected set of symbols. (2) See *bus layout, ring layout, tree layout,* and *star layout*. See also *layout algorithm*.

polling. (1) On a multipoint connection or a point-to-point connection, the process whereby data stations are invited, one at a time, to transmit. (I) (2) Interrogation of devices for such purposes as to avoid contention, to determine operational status, or to determine readiness to send or receive data. (A)

pop-up menu. In the AIXwindows program, a type of MenuPane widget that appears as the result of some user action, usually clicking a mouse button, and then disappears when the action is completed.

port. (1) An access point for data entry or exit. (2) A connector on a device to which cables for other devices, such as display stations and printers, are attached. (3) In TCP/IP, a 16-bit number used to communicate between TCP and a higher-level protocol or application. Some protocols, such as the File Transfer Protocol (FTP) and the Simple Mail Transfer Protocol (SMTP), use the same port number in all TCP/IP implementations. (4) The representation of a physical connection to the link hardware. A port is sometimes referred to as an adapter. There may be one or more ports controlled by a single DLC process.

postmaster. (1) A process (daemon) that directs network management information between multiple applications and agents running concurrently. The postmaster determines the route by using specified addresses or a routing table that is configured in the object registration service. (2) See also *pmd daemon* and *object registration file (ORS)*.

process ID. A unique number that is assigned by the AIX Operating System to each program that is running. (R)

protocol. (1) A set of semantic and syntactic rules that determine the behavior of functional units in achieving communication. (I) (2) In Open Systems Interconnection architecture, a set of semantic and syntactic rules that determine the behavior of entities in the same layer in performing communication functions. (T)

proxy agent. A “translator” routine that manages an object and converts its communications defined by one protocol.

pull-down menu. In the AIXwindows program, a type of MenuPane widget that gives the appearance of being pulled down from a MenuBar widget as the result of a user action, usually, clicking a mouse button.

R

real time. (1) In Open Systems Interconnection architecture, pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process. This term is also used to describe systems operating in conversational mode and processes that can be influenced by human intervention while they are in progress. (I) (A) (2) In Open Systems Interconnection architecture, pertaining to an application such as a process control system or a computer-assisted instruction system in which response to input is fast enough to affect subsequent input.

Recommendation X.25. An International Telegraph and Telephone Consultative Committee (CCITT) recommendation for the interface between data terminal equipment and packet-switched data networks.

record. (1) In programming languages, an aggregate that consists of data objects, possibly with different attributes, that usually have identifiers attached to them. In some programming languages, records are called structures. (I) (2) A set of data treated as a unit. (TC97) (3) A set of one or more related data items grouped for processing.

recording filter. In the NetView program, the function that determines which events, statistics, and alerts are stored on a database.

registration file. See *application registration file*.

remote. (1) Pertaining to a system, program, or device that is accessed through a telecommunication line. (2) A device that does not use the same protocol and is, therefore, unknown. (3) Contrast with *local.*

resource. Any facility of the computing system or operating system required by a job or task, and including main storage, input/output devices, the processing unit, data sets, and control or processing programs.

response. (1) In data communications, a reply represented in the control field of a response frame. It advises the primary or combined station of the action taken by the secondary or other combined station to one or more commands. (2) See also *command*.

ring. A network configuration in which devices are connected by unidirectional transmission links to form a closed path.

ring layout. (1) A layout algorithm in which symbols are arranged in a ring on a submap. A ring layout is one of several layout algorithms that are available in the AIX SystemView NetView/6000 windows tool. (2) (3) See *bus layout, point-to-point layout, row/column layout, tree layout,* and *star layout*. See also *layout algorithm*.

RISC. Reduced instruction-set computer.

root-level submap. Contains the highest level of the submap hierarchy. Multiple networks can be placed within the root level submap.

root user. See *superuser authority.*

route. An ordered sequence of nodes and transmission groups (TGs) that represents a path from an origin node to a destination node traversed by the traffic exchanged between them.

router. See Internet router.

routing. (1) The process of determining the path to be used for transmission of a message over a network. (T) (2) The assignment of the path (route) by which a message will reach a destination on the network. (3) In X.25 communications, the process by which a packet gets to the intended user. (R)

row/column layout. (1) A layout algorithm in which symbols are arranged in rows and columns on a submap. (2) See *bus layout, ring layout, point-to-point layout, tree layout,* and *star layout*. See also *layout algorithm*.

S

scaling. (1) The way a submap is presented within the submap window. Scaling or zooming is available for each submap. In scaling, the AIX SystemView NetView/6000 program scales symbols on the map to the size of the submap window. When windows are resized, the symbols and background graphics change size to reflect the new window size. Scaling is the default setting for submaps. (2) See *zoom*.

screen. (1) In the AIX extended curses library, a window that is as large as the display screen of the workstation. (2) Deprecated term for display panel.

scroll. To move a display image vertically or horizontally to view data that cannot be observed within a single display screen.

seed file. In the AIX SystemView NetView/6000 program, a file that contains a list of nodes within an administrative domain, which the automatic discovery function uses to accelerate the generation of the network topology map.

segment. (1) A group of display elements. (2) A contiguous area of virtual storage allocated to a job or system task. A program segment can be run by itself, even if the whole program is not in main storage. (3) A portion of a computer program that may be executed without the entire program being resident in main storage. (4) In AIX Enhanced X Windows, one or more lines that are drawn but not necessarily connected at the end points. (5) In the IBM Token-Ring Network, a section of cable between components or devices on the network. A segment may consist of a single patch cable, multiple patch cables connected together, or a combination of building cable and patch cables connected together.

segment-level submap. Represents the topology of a segment of a network. A segment submap contains network nodes and connectors.

select. (1) In the AIX Operating System, to choose a button on the display screen. (2) To place the cursor on an object (name or command) and press a button on the mouse or the appropriate key on the keyboard.

selection list. A list of selected objects for the open map or open snapshot. The selection list provides a list of objects on which various operations can be invoked by users and applications. As objects are selected or deselected, the selection list is immediately updated. The selection list will display the objects' selection names. When a map and snapshot are open concurrently, each will have an independent list of selected objects.

selection name. (1) The unique name of an object that is used in a selection list. The AIX SystemView NetView/6000 program identifies an object by its selection name. An object can be assigned a selection name through the Add Object dialog box. (2) See also *object* and *object attribute*.

server. (1) In the AIX Operating System, an application program that usually runs in the background and is controlled by the system program controller. (2) In Enhanced X Windows, provides the basic windowing mechanism. It handles IPC connections from clients, demultiplexes graphics requests onto screens, and multiplexes input back to clients. (3) See also *client*.

service access point. In multiple protocol topology, a service access point indicates correlation between protocols. This is the service that an entity, such as, the internet protocol uses or provides another entity. For example, the internet protocol uses the services of a token-ring adapter. The service access point, in this case, is the name by which the internet protocol knows the adapter that is the token-ring address.

shared. Pertaining to the availability of a resource for more than one use at the same time.

shared submap. A submap on which multiple applications manage objects on the application plane. Shared submaps allow applications to cooperatively contribute information to the same submap.

shell script. A synonym for shell procedure.

simple connection. In multiple topology, a simple connections represents connectivity as seen from one end point. This may contain specific information about one end point of the connection.

Simple Network Management Protocol (SNMP). A protocol running above the User Datagram Protocol (UDP) used to exchange network management information.

SMIT. System Management Interface Tool

SNA. Systems Network Architecture.

SNMP. Simple Network Management Protocol.

SP. Service point.

star layout. (1) A layout algorithm where the symbols are arranged in a star. (2) See *bus layout, point-to-point layout, ring layout, tree layout,* and *row/column layout*. See also *layout algorithm*.

startup file. A file that contains information about the ordered sequence of network management processes, such as daemons and agents. The startup sequence is listed in the /usr/OV/conf/ovsuf file.

static. (1) In programming languages, pertaining to properties that can be established before execution of a program; for example, the length of a fixed length variable is static. (T) (2) In AIX SystemView NetView/6000, static workspace contains only certain events. The static workspace is not updated. (3) Pertaining to an operation that occurs at a predetermined or fixed time. (4) Contrast with *dynamic*.

station. An input or output point of a system that uses telecommunications facilities; for example, one or more systems, computers, terminals, devices, and associated programs at a particular location that can send or receive data over a telecommunication line.

status. (1) The current condition or state of a program or device. (R) (2) In the AIX SystemView NetView/6000 program, the condition of a node or portion of a network as represented by the color of a symbol on a submap.

status propagation. (1) The changing of the status of compound objects based on the propagation rules of the objects contained entirely within it. (2) See also *propagate most critical and propagate threshold*.

status source. (1) The source of the status being collected, propagated, or displayed. The source of symbol status may come from:

- An application that sets status on a specific symbol, which is called Symbol Status Source.
- An application that sets the same status on all symbols of a given object, which is called Object Status Source.
- Compound Status.

(2) See also *object status source, symbol status source,* and *compound status*.

submap. (1) A particular view of some aspect of a network that displays symbols that represent objects. Some symbols may explode into other submaps, usually having a more detailed view than their parent submap. The application that creates a submap determines what part of the network the submap displays.

(2) See also *root-level submap, internet-level submap, node-level submap,* and *segment-level submap*.

submap window. A submap window contains an AIX SystemView NetView/6000 menu bar, a submap viewing area, a status line, and a button box. You can display multiple submap windows of an open map and an open snapshot at any given time.

subnet. (1) In TCP/IP, a part of a network that is identified by a portion of the Internet address. (2) In the AIX Operating System, synonym for subnetwork.

subnetwork. (1) In the AIX Operating System, one of a group of multiple logical network divisions of another network, such as can be created by the Transmission Control Protocol/Internet Protocol (TCP/IP) interface program. (2) Any group of nodes that have a set of common characteristics, such as the same network ID.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (T)

superuser authority. (1) In the AIX Operating System, the unrestricted authority to access and modify any part of the operating system, usually associated with the user who manages the system. (R) (2) See *root user*.

symbol. (1) In the AIX SystemView NetView/6000 program, a picture or icon that represents an object. Each symbol has an outside and inside component.

- The outside component differentiates the object classes.
- The inside component differentiates the objects within the class. (2) See also *object class* and *symbol class*.

symbol class. (1) A collection of symbols that have the same or similar properties. A symbol class is represented by the shape of the symbol. Each symbol subclass in a given class contains the same shape. Each symbol class has a unique set of subclasses associated with it. Applications may register additional symbol classes. Some of the registered symbol classes provided with the AIX SystemView NetView/6000 program include:

- Computer
- Connector
- Device
- Software
- Location
- Cards

You can view all the registered classes from the Display Legend panel, or from the Add Object Palette. (2) See also *symbol subclass* and *symbol type*.

symbol graphic. A graphic, such as a picture of a workstation, used to distinguish symbol subclasses.

symbol label. A name assigned to a symbol to distinguish it from other symbols in a submap. A symbol label may be displayed below the symbol. Symbol labels are optional.

symbol status source. (1) A setting that allows applications to set the status of a particular symbol on a submap. Other symbols representing the same object are not affected by the symbol's status. Symbol Status Source is useful for propagating status through an application and for graphically displaying status specific to the semantics of a particular submap. (2) See also *object status source*.

symbol subclass. (1) A set of symbols that is in a given symbol class. A particular subclass in a given class defines the type of the symbol. For example, in the symbol class called Computer, the subclasses consist of PC, workstation, mini, and mainframe. All symbols in the subclass of the same class contain the same outline (shape). The AIX SystemView NetView/6000 program displays the symbol subclass as the graphic inside the outer shape of the symbol. (2) See also *symbol class* and *symbol type*.

symbol type. (1) The symbol type consists of the symbol class and the symbol subclass. A specific symbol type is defined by the concatenation of a symbol class and a symbol subclass within that class. The symbol class is identified on submaps by the outer shape of the symbol and the symbol subclass by the graphic inside the shape. (2) See also *symbol class* and *symbol subclass*.

SystemView. The IBM systems management strategy for planning, coordinating, and operating open, heterogeneous, enterprise-wide information systems.

T

target objects. (1) Objects targeted for some action by an application. When the behavior of a symbol is set as executable, an application is chosen to perform an action on a set of objects, called target objects. The objects that are targeted by the application can also be chosen. Each executable symbol may have its own set of target objects. (2) See also *executable symbol*.

task. In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. (I) (A)

task index. (1) An index that provides online help entries for a variety of tasks that are available in the AIX SystemView NetView/6000 program and applications that are integrated with the AIX SystemView

NetView/6000 program. The Task Index can be accessed from the Help menu. (2) See also *help menu*.

TCP. Transmission Control Protocol.

TCP/IP. Transmission Control Protocol/Internet Protocol.

telnet. An application protocol offering virtual terminal service in the Internet suite of protocols. With the telnet service, a remote system can be connected to a local system.

terminal. (1) A device, usually equipped with a keyboard and a display device, capable of sending and receiving information over a communications line. (2) See also *workstation*. (R)

threshold. In the AIX SystemView NetView/6000 program, a setting that specifies the maximum value a statistic can reach before notification that the limit was exceeded. For example, when a monitored MIB value has exceeded the threshold, SNMPCollect generates a threshold event.

token. (1) In a local area network, the symbol of authority passed successively from one data station to another to indicate the station temporarily in control of the transmission medium. Each data station has an opportunity to acquire and use the token to control the medium. A token is a particular message or bit pattern that signifies permission to transmit. (T) (2) A sequence of bits passed from one device to another along the token ring. When the token has data appended to it, it becomes a frame.

token ring. A network with a ring topology that passes tokens from one attaching device to another. For example, the IBM Token-Ring Network.

tool palette. (1) A component of the graphical interface that enables the operator to open application instances by using the mouse to drag-and-drop the icons that represent the application. (2) See also *control desk*.

topology. In the AIX SystemView NetView/6000 program, the geographical, logical layout of a network. The way in which the nodes in a network are interconnected.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that supports peer-to-peer connectivity functions for both local and wide-area networks.

trap. An unsolicited event generated by an agent and forwarded to a manager. Traps inform the manager of changes that occur in the network.

tree layout. (1) A layout algorithm that displays a tree configuration and shows symbols arranged in a hierarchical tree structure. (2) See *point-to-point layout, ring layout, bus layout, row/column layout,* and *star layout*.

U

underlying arc. In multiple topology, an underlying arc represents a lower-layer end point that is independently connected and is being used by a higher-layer end point connection.

underlying connection. In multiple topology, an underlying connection represents lower-layer connectivity that is used by higher-layer connectivity. For example, two IP hosts that use a physical connection for transport. The physical connection that transports data between the hosts is the underlying connection.

UDP. User Datagram Protocol.

UNIX Operating System. An operating system developed by Bell Laboratories that features multiprogramming in a multiuser environment. The UNIX Operating System was originally developed for use on minicomputers but has been adapted for mainframes and microcomputers.

Note: The AIX Operating System is IBM's implementation of the UNIX operating system. See *AIX*.

unknown status. (1) The status of an object that is not yet known or does not actually exist in the network. The default icon symbol color for unknown status is Blue. The default connection symbol color is Black. (2) See *critical status, normal status, unknown status, unmanaged status,* and *status*.

unmanaged object. (1) An object that is not actively managed. An unmanaged object displays status as Unmanaged. It does not display active status (normal, marginal, critical). Unmanaged objects do not display compound status nor do they contribute to compound status. Objects can be kept in an unmanaged state if they are not of interest. An object may be toggled between a managed and unmanaged state. (2) See *managed object* and *unmanaged status*.

unmanaged status. (1) The status that indicates that an object is unmanaged. The default icon symbol color displayed to indicate unmanaged status is Wheat. The default connection symbol color displayed is Black. (2) See *critical status, normal status, compound status, unknown status,* and *status*.

user. Any person or anything that may issue commands and messages to or receive commands and messages from the information processing system. (T)

User Datagram Protocol (UDP). (1) In IP, a packet-level protocol built directly on the Internet protocol layer. UDP is used under SNMP for application-to-application programs between IP host systems. (2) In AIX SystemView NetView/6000, the TCP/IP standard datagram protocol.

user plane. (1) The plane on a submap in which the AIX SystemView NetView/6000 program displays symbols that are not managed by any application. You can visually distinguish symbols in the user plane from symbols in the application plane. Symbols in the user plane are displayed with a shadow, which makes the symbol appear higher than symbols in the application plane. (2) See *application plane*.

V

value. (1) A specific occurrence of an attribute, for example, "blue" for the attribute color. (TC97) (2) A quantity assigned to a constant, a variable, a parameter, or a symbol.

variable. (1) A name used to represent a data item whose value can change while the program is running. (2) In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. (3) A quantity that can assume any of a given set of values. (A)

vertex. In multiple topology, the lowest point that can be defined. A vertex contains physical and logical interfaces. Logical interfaces are protocols, such as IP. Physical interfaces are adapters, such as token-ring, ethernet, and fiber distributed data interface (FDDI).

view. See *submap*.

viewing filter. In the NetView program, the function that allows a user to select the alert data to be displayed on a terminal. All other stored data is blocked.

view menu. An action bar menu that provides options for changing the way symbols are displayed on individual or all submaps.

W

WAN. Wide area network.

wide-area network (WAN). (1) A network that provides communication services to a geographical area larger than those serviced by a local area network or a metropolitan area network, and that may use or provide public communication facilities. (T) (2) A data communications network designed to serve an area of hundreds or thousands of miles; for example, public and private packet-switching networks, and national telephone networks. (3) Contrast with *local area network*.

widget. (1) In the AIX Operating System, a graphic device that can receive input from the keyboard or mouse and communicate with an application or with another widget by means of a callback. Every widget is a member of only one class and always has a window associated with it. (2) The fundamental data type of the AIX Enhanced X WindowsToolkit. (3) An object that provides a user-interface abstraction; for example, a Scrollbar widget. It is the combination of an AIX Enhanced X Windows window (or subwindow) and its associated semantics. A widget implements procedures through its widget class structure.

wildcard character. Synonym for pattern-matching character.

window. A portion of a visual display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A)

workspace. (1) That portion of main storage that is used by a computer program to store the objects a user creates. (2) Acts as a card holder for an event display application. Most event-display-application functions are applied on workspaces as report generation, search, order, and append.

workstation. (1) A functional unit at which a user works. A workstation often has some processing capability. (T) (2) One or more programmable or nonprogrammable devices that allow a user to do work. (3) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

X

X Window System. (1) A network-transparent windowing system developed by the Massachusetts Institute of Technology. It is the basis for Enhanced X-Windows, which runs on the AIX Operating System. (2) See also *Enhanced X-Windows Toolkit*.

X.25. (1) A CCITT Recommendation that defines the physical level (physical layer), link level (data link layer), and packet level (network layer) of the OSI reference model. An X.25 network is an interface between data terminal equipment (DTE) and data circuit-terminating equipment (DCE) operating in the packet mode, and connected to public data networks by dedicated circuits. X.25 networks use the connection-mode network service. (2) See *recommendation X.25*.

X11. (1) X Window System, Version 11. (2) See also *X Window System*.

Z

zoom. (1) To progressively increase or decrease a part of an image on a screen or in a window. (2) See also *scaling* and *zoom factor*.

zoom factor. (1) Used to determine the magnification of a submap. You can choose a zoom factor up to ten times the normal view. (2) See *scaling* and *zoom*.

Bibliography

NetView for AIX Publications

The following paragraphs briefly describe the publications for Version 4 of the NetView for AIX program:

NetView for AIX Concepts: A General Information Manual (GC31-8160)

This book provides an overview of the NetView for AIX program that business executives can use to evaluate the product. System planners can also use this information to learn how NetView for AIX manages heterogeneous networks.

NetView for AIX Database Guide (SC31-8167)

This book provides information for system administrators and database administrators to configure the NetView for AIX program to work with the following relational database management systems: DB2/6000, INFORMIX, INGRES, ORACLE, and SYBASE. This book also describes how to transfer IP topology, trapdlog, and snmpCollect data to the relational database and how to manipulate the data.

NetView for AIX Installation and Configuration (SC31-8163)

This book provides installation and configuration steps for the system programmer who will install and configure the NetView for AIX program.

NetView for AIX User's Guide for Beginners (SC31-8158)

This book contains “how-to” information that provides network operators the help they need to get acquainted with NetView for AIX and accomplish some basic networking tasks. It is written for the user who is unfamiliar with the NetView for AIX program.

NetView for AIX Administrator's Guide (SC31-8168)

This book explains network management principles and describes how the NetView for AIX program's components work together. It is for the advanced user. Most of the tasks require root authority. This book includes tasks such as customizing the graphical interface, filtering events, configuring events, and managing network performance and configuration.

NetView for AIX Administrator's Reference (SC31-8169)

This book contains reference information for commands, daemons, and files. It is used primarily when performing administrative tasks.

NetView for AIX Diagnosis Guide (SC31-8162)

This book is intended to help you classify and resolve problems related to the operation of the NetView for AIX program.

NetView for AIX Application Interface Style Guide (SC31-6240)

This book provides guidelines for system programmers who develop applications that will be integrated with the NetView for AIX program.

NetView for AIX Programmer's Guide (SC31-8164)

This book provides information for programmers about creating network management applications. This book also contains information about the NetView for AIX program server, commands, function calls, and object classes.

NetView for AIX Programmer's Reference (SC31-8165)

This book is intended for programmers and contains reference information about the NetView for AIX program and its server, commands, function calls, and object classes.

NetView for AIX and the Host Connection (SC31-8161)

This book provides information for System/390 and NetView users who want to manage TCP/IP and SNA networks.

Quick Reference Card (SX75-0113)

This summary provides a brief description of each NetView for AIX daemon. The card also lists the menu items and the submenu items below them.

In addition to these printed books, online documentation of the NetView for AIX library is available. An online Help Index is also available from the NetView for AIX Help pull-down window. The Help Index provides dialog box help and task help.

IBM RISC System/6000 Publications

In addition to the NetView for AIX documentation, the following publications may also be helpful to users:

AIX Quick Reference (SC23-2401)

Task Index and Glossary for IBM RISC System/6000 (GC23-2201)

IBM RISC System/6000 Problem Solving Guide (SC23-2204)

AIX Communications Concepts and Procedures for IBM RISC System/6000 (GC23-2203)

AIX Commands Reference for IBM RISC System/6000 (GC23-2366, GC23-2367, GC23-2376, GC23-2393)

AIX Files Reference for IBM RISC System/6000 (GC23-2200)

NetView Publications

The following list contains selected NetView Version 2 Release 3 publications:

NetView Administration Reference (SC31-6128)

NetView At a Glance (GC31-7016)

NetView Automation Planning (SC31-6141)

NetView Customization Guide (SC31-6132)

NetView Installation and Administration Guide (MVS: SC31-6125) (VM: SC31-6182) (VSE: SC31-6182)

NetView Operation (SC31-6127)

NetView Problem Determination and Diagnosis (LY43-0014)

NetView Resource Alerts Reference (SC31-6136)

NetView Samples (MVS: SC31-6126) (VM: SC31-6183) (VSE: SC31-6184)

The following list contains selected NetView Version 2 Release 4 publications:

NetView Administration Reference (SC31-7080)

NetView Automation Planning (SC31-7082)

NetView Customization Guide (SC31-7091)

NetView General Information (GC31-7098)

NetView Installation and Administration Facility/2 Guide (SC31-7099)

NetView Installation and Administration Guide (SC31-7084)

NetView Operation (SC31-7066)

NetView Problem Determination and Diagnosis (LY43-0101)

NetView Resource Alerts Reference (SC31-7097)

TCP/IP Publications for AIX (RS/6000, PS/2, RT, 370)

The following list shows the books available for TCP/IP in the AIX Operating System library:

AIX Operating System TCP/IP User's Guide (SC23-2309)

AIX PS/2 TCP/IP User's Guide (SC23-2047)

TCP/IP for IBM X-Windows on DOS (SC23-2349)

AIX SNA Services/6000 Publications

The following list of publications are for use with the AIX Operating System:

AIX SNA Server/6000 User's Guide (SC31-7002)

AIX SNA Server/6000 Configuration Reference (SC31-7014)

AIX SNA Server/6000 Transaction Program (SC31-7003)

Internet Request for Comments (RFCs)

The following documents describe Internet standards supported by the NetView for AIX program. Copies of these documents are shipped on the AIX SystemView NetView/6000 product installation media. They are installed in the /usr/OV/doc directory.

RFC 1095: The Common Management Services and Protocol over TCP/IP (CMOT)

RFC 1155: Structure and Identification of Management Information for TCP/IP-Based Internets

RFC 1157: Simple Network Management Protocol (SNMP)

RFC 1187: Bulk Table Retrieval with the SNMP

RFC 1189: The Common Management Information Services and Protocols for the Internet (CMOT and CMIP)

RFC 1212: Concise MIB Definitions

RFC 1213: Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II

RFC 1215: Convention for Defining Traps for Use with the SNMP

RFC 1229: Extensions to the Generic-Interface MIB

RFC 1230: IEEE 802.4 Token Bus MIB

RFC 1231: IEEE 802.5 Token Bus MIB

RFC 1232: Definitions of Managed Objects for the DS1 Interface Type

RFC 1233: Definitions of Managed Objects for the DS3 Interface Type

RFC 1239: Reassignment of Experimental MIBs to Standard MIBs

RFC 1243: AppleTalk Management Information Base

RFC 1253: OSPF Version 2 Management Information Base

RFC 1269: Definitions of Managed Objects for the Border Gateway Protocol (Version 3)

RFC 1271: Remote Network Monitoring Management Information Base

RFC 1284: Definitions of Managed Objects for the Ethernet-like Interface Types

RFC 1285: FDDI Management Information Base

RFC 1286: Definitions of Managed Objects for Bridges

RFC 1289: DECnet Phase IV MIB Extensions

RFC 1304: Definition of Managed Objects for the SIP Interface Type

RFC 1315: Management Information Base for Frame Relay DTEs

RFC 1316: Definitions of Managed Objects for Character Stream Devices

RFC 1317: Definitions of Managed Objects for RS-232-like Hardware Devices

RFC 1318: Definitions of Managed Objects for Parallel-printer-like Hardware Devices

RFC 1450: Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1452: Coexistence between Version 1 and Version 2 of the Internet-Standard Network Management Framework

Related Publications

The following publications are closely related to or referenced by the NetView for AIX Library:

AIX Trouble Ticket/6000 Publications

For information about the AIX Trouble Ticket/6000 program, consult the following publications:

AIX Trouble Ticket/6000 Brochure (GC31-7161)

AIX Trouble Ticket/6000 User's Guide (SC31-7162)

Service Point Publication

AIX NetView Service Point Installation, Operation, and Programming Guide (SC31-6120)

Other IBM TCP/IP Publications

The following list shows other available IBM TCP/IP publications:

Introducing IBM Transmission Control Protocol/Internet Protocol Products for OS/2, VM, and MVS (GC31-6080)

IBM TCP/IP Version 2 for VM and MVS: Diagnosis Guide (LY43-0013)

MVS/DFP Version 3 Release 3: Using the Network File System Server (SC26-4732)

SNMP Information

You can use the following sources for detailed SNMP information:

The Simple Book, M.T. Rose, Prentice-Hall, 1991 (ISBN 0-13-812611-9)

The *Windows SNMP Manager API Specification*, the *WinSNMP/MIB API Specification*, and other information on Windows SNMP are available through anonymous FTP from the host sunsite.unc.edu under the directory path /pub/micro/pc-stuff/ms-windows/WinSNMP

These Internet standards provide SNMP information:

RFC 1901: Introduction to Community-based SNMPv2

RFC 1902: Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1903: Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1904: Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1905: Protocol Operation for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1906: Transport Mapping for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1907: Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1908: Coexistence between Version 1 and Version 2 of Internet-standard Network Management Framework

RFC 1909: An Administrative Infrastructure for SNMPv2 (SNMPv2USEC)

RFC 1910: User-based Security Model for SNMPv2 (SNMPv2USEC)

X Window System Publications

The following list shows selected X Window System publications:

Introduction to the X Window System, Oliver Jones, Prentice-Hall, 1988 (ISBN 0-13-499997)

X Window System Technical Reference, Steven Mikes, Addison-Wesley, 1990 (ISBN 0-201-52370)

X Window System: Programming and Applications with Xt, Douglas A. Young, Prentice-Hall, 1989 (ISBN 0-13-972167)

X Window System: Programming and Applications with Xt, OSF/Motif Edition, Douglas A. Young, Prentice-Hall, 1990 (ISBN 0-13-497074)

X/Open Specification

For information about the X/Open OSI-Abstract-Data Manipulation (XOM) application programming interface (API), consult the following X/Open documents:

X/Open OSI-Abstract-Data Manipulation (XOM) API, CAE Specification

X/Open Preliminary Specification. Systems Management: GDMO to XOM Translation Algorithm

OSF/Motif Publications

The following list contains selected OSF/Motif publications:

OSF/Motif Series (5 volumes), Open Software Foundation, Prentice Hall, Inc. 1990

OSF/Motif Application Environment Specifications, (AES) (ISBN 0-13-640483-9)

OSF/Motif Programmer's Guide (ISBN 0-13-640509-6)

OSF/Motif Programmer's Reference, (ISBN 0-13-640517-7)

OSF/Motif Style Guide (ISBN 0-13-640491-X)

OSF/Motif User's Guide, (ISBN 0-13-640525-8)

ISO/IEC Standards

For information about the ISO/IEC standards on which the NetView for AIX program is based, refer to the following publications:

ISO IS 7498-4, Open Systems Interconnection–Basic Reference Model–Part 4: Management Framework

ISO 8824, Open Systems Interconnection–Specification of Abstract Syntax Notation One (ASN.1)

ISO 8825, Open Systems Interconnection– Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)

ISO IS 9595, Common Management Information–Service Definition

ISO IS 9596-1, Common Management Information–Protocol Specification

ISO DIS 9899, Information Processing–Programming Language C

ISO 10040, Systems Management Overview

The ISO/IEC standards can be obtained from the following address:

OMNICOM
243 Church St. NW
Vienna, VA 22180-4434

(800) OMNICOM
(703) 281-1135
(703) 281-1505 (FAX)

Index

Communicating Your Comments to IBM

NetView for AIX
Programmer's Guide
Version 4

Publication No. SC31-8164-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:

  United States and Canada: **1-800-227-5088**
- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**
  - Internet: **USIB2HPD@VNET.IBM.COM**

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Help us help you!

NetView for AIX
Programmer's Guide
Version 4

Publication No. SC31-8164-00

We hope you find this publication useful, readable and technically accurate, but only you can tell us! Your comments and suggestions will help us improve our technical publications. Please take a few minutes to let us know what you think by completing this form.

Overall, how satisfied are you with the information in this book?	Satisfied	Dissatisfied
	☐	☐

How satisfied are you that the information in this book is:	Satisfied	Dissatisfied
Accurate	☐	☐
Complete	☐	☐
Easy to find	☐	☐
Easy to understand	☐	☐
Well organized	☐	☐
Applicable to your task	☐	☐

Specific Comments or Problems:

Please tell us how we can improve this book:

Thank you for your response. When you send information to IBM, you grant IBM the right to use or distribute the information without incurring any obligation to you. You of course retain the right to use the information in any way you choose.

Your Internet Address: ______________________

Name

Address

Company or Organization

Phone No.

Help us help you!
SC31-8164-00

IBM®

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Information Development
Department CGMD
International Business Machines Corporation
PO BOX 12195
RESEARCH TRIANGLE PARK NC 27709-9990

Fold and Tape **Please do not staple** Fold and Tape

SC31-8164-00

Cut or Fold
Along Line

Program Number: 5765-527

Printed in U.S.A.

SC31-8164-00

```
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2330 OF '.EDF#CV'
DSMMOM397I '.EDF#CV' WAS IMBEDDED AT LINE 190 OF '.EDF#FCV7'
DSMMOM397I '.EDF#FCV7' WAS IMBEDDED AT LINE 330 OF '.EDFCOVER'
DSMMOM397I '.EDFCOVER' WAS IMBEDDED AT LINE 49 OF 'LBVL0MST'
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2 OF 'LBVL0C2'
DSMMOM397I 'LBVL0C2' WAS IMBEDDED AT LINE 187 OF 'EDFPRF40'
DSMBEG323I STARTING PASS 2 OF 4.
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2330 OF '.EDF#CV'
DSMMOM397I '.EDF#CV' WAS IMBEDDED AT LINE 190 OF '.EDF#FCV7'
DSMMOM397I '.EDF#FCV7' WAS IMBEDDED AT LINE 330 OF '.EDFCOVER'
DSMMOM397I '.EDFCOVER' WAS IMBEDDED AT LINE 49 OF 'LBVL0MST'
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2 OF 'LBVL0C2'
DSMMOM397I 'LBVL0C2' WAS IMBEDDED AT LINE 187 OF 'EDFPRF40'
DSMBEG323I STARTING PASS 3 OF 4.
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2330 OF '.EDF#CV'
DSMMOM397I '.EDF#CV' WAS IMBEDDED AT LINE 190 OF '.EDF#FCV7'
DSMMOM397I '.EDF#FCV7' WAS IMBEDDED AT LINE 330 OF '.EDFCOVER'
DSMMOM397I '.EDFCOVER' WAS IMBEDDED AT LINE 49 OF 'LBVL0MST'
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2 OF 'LBVL0C2'
DSMMOM397I 'LBVL0C2' WAS IMBEDDED AT LINE 187 OF 'EDFPRF40'
DSMBEG323I STARTING PASS 4 OF 4.
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2330 OF '.EDF#CV'
DSMMOM397I '.EDF#CV' WAS IMBEDDED AT LINE 190 OF '.EDF#FCV7'
DSMMOM397I '.EDF#FCV7' WAS IMBEDDED AT LINE 330 OF '.EDFCOVER'
DSMMOM397I '.EDFCOVER' WAS IMBEDDED AT LINE 49 OF 'LBVL0MST'
DSMKPO653E POSTSCRIPT FILE '@E@P@S' NOT FOUND.
DSMMOM395I '.EDFPO' LINE 70: .po @E@P@S
DSMMOM397I '.EDFPO' WAS IMBEDDED AT LINE 910 OF '.EDFAWRK'
DSMMOM397I '.EDFAWRK' WAS IMBEDDED AT LINE 2 OF 'LBVL0C2'
DSMMOM397I 'LBVL0C2' WAS IMBEDDED AT LINE 187 OF 'EDFPRF40'
```