Developer's Reference Guide

for the Apple Network Server

Contents

Part III AppleTalk Programming Interfaces

11 AppleTalk Programming Interfaces

Part IV Manual Pages

Figures and Tables

Preface

This guide describes the changes that affect developers for AIX and AppleTalk services on the Network Server. This guide is not intended to replace the existing IBM AIX documentation; it simply supplements the IBM documentation and highlights the differences you may encounter when developing for AIX for the Network Server.

Who should use this guide

Programmers should read this guide who

- need to create Mac OS client applications for the Network Server
- need to write or modify device drivers for the Network Server
- use the AppleTalk API
- need to consult the reference pages for new commands

Conventions used in this guide

This guide follows specific conventions to convey information more clearly. For example, words that require special emphasis appear in specific fonts or font styles. The following sections describe the conventions used in this guide.

The Courier font

Throughout this manual, words that appear on the screen or that you must type exactly as shown are in the Courier font. For example, suppose you see this instruction:

Type `date` on the command line and press RETURN.

The word `date` is in the Courier font to indicate that you must type it.

Suppose you then read this explanation:

After you press RETURN, information such as this appears on the screen:

```
Tues Oct 17 17:04:00 PDT 1989
```

In this case, Courier is used to represent the text that appears on the screen.

Italics

When shown in text, commands often contain “placeholder” words or characters that appear in italics. These placeholders represent parts of a command for which you substitute different values when you actually enter the command. For example, in the sample command

`cat` *file*

the word *file* is a placeholder for the name of a file you want to display. If you wanted to display the contents of a file named `Elvis`, you would issue the `cat` command, typing the word `Elvis` in place of *filename*. In other words, you would enter

```
cat Elvis
```

Sometimes italic placeholders are used in other contexts—for example, to represent text that appears on the screen or to represent the value of a field in a file. Here is a sample prompt that might appear on your screen:

`Apple Computer, Inc. (`*hostname*`)`

`login:`

In this prompt, the word *hostname* is a placeholder for the name of the computer to which you can log in.

Command notation

This guide uses special notation to present commands. This notation is designed to reflect the syntax of the command—that is, to indicate how to enter the command so that its structure is legal and its parts are interpreted properly.

Here is a description of each element of the command notation.

Element	Description
command	The command name. This element appears in the Courier font, as explained earlier.
option	A character or group of characters that modifies the command. Most options have the form `-`*option*, where *option* is a letter representing an option. Most commands have one or more options.
argument	A value that modifies the behavior of a command, typically the name of an object that the command acts upon.
[ ]	Brackets used to enclose an optional item—that is, an item that is not essential for execution of the command.
...	Ellipsis points are used to indicate that you can enter the argument preceding the ellipsis points more than once.

A typical command line comprises the command name, followed by options and arguments. For example, the `wc` command would look like this:

`wc [-c][-l][-w]` *file*...

In this example, `wc` is the command, `-c`, `-l`, and `-w` are options of which you can specify zero or more; *file* indicates that an argument consisting of a filename is required; and the ellipsis points (...) indicate that you can specify more than one filename argument. Brackets and ellipses are *not* to be typed. Also, note that each command element is separated from the next element by a space.

To count the words in a file named `Priscilla`, you would use the `-w` option and replace the placeholder *file* with that filename. The command you enter would look like this:

```
wc -w Priscilla
```

Section 1 Developing Client/Server Applications for the Network Server Using Apple Protocols

This section describes how to develop client/server applications for the Network Server, which let users access AIX on Network Servers from Macintosh computers. These client/server applications have two components: a Network Server component that acts as a server and a Macintosh component that acts as a client. You develop the two components separately and provide for the exchange of information between them.

This section details how these client/server applications work, how to develop both the Macintosh and Network Server components, and how to exchange messages between the two components.

This section is intended for programmers and developers who want to create applications with a Macintosh interface and a Network Server back end.

To get the most out of this section, you need to understand program development in both the Macintosh and AIX environments, and you need to be familiar with Apple events.

This section presents sample routines using Metrowerks CodeWarrior and the Macintosh Programmer's Workshop (MPW). The sample code listings are shown in C and C++.

This section contains the following chapters:

- Chapter 1, "Overview of Client/Server Applications for the Network Server," describes how the components of client/server applications work and discusses the protocols they use.
- Chapter 2, "Developing the Macintosh Component," describes how to establish communication with the Network Server, use Apple events, and send "heartbeat" Apple events.
- Chapter 3, "Developing the Network Server Component," covers include files and libraries; it also describes using Apple events, sending heartbeat Apple events, and debugging the Network Server component.
- Chapter 4, "A Sample Application," describes the files used in both the Macintosh and Network Server components of the Status Demo AppleTalk Services application, which is provided with AIX for the Network Server.

For more information

The following technical books, documents, and resources provide additional information about developing client/server applications, AppleTalk and Apple events, and UNIX and Macintosh programming.

- Macintosh Programmer's Workshop (MPW)

  The MPW development environment includes these books: *Macintosh Programmer's Workshop Development Environment*, Volume 1;*Macintosh Programmer's Workshop Development Environment*, Volume 2; *MPW Pascal*; *Macintosh Programmer's Workshop C*. These books are available from APDA. (See the ordering information later in this Preface.)

- MacApp development environment

  To use the MacApp development environment, consult the *Programmer's Section to MacApp* (available from APDA).

- AIX development environment

  For information about the available compilers, debuggers, and development tools, see the AIX documentation.

- AppleTalk networks

  For additional information about Macintosh networking and communication, see *Inside AppleTalk*, second edition. This technical reference describes the AppleTalk network system protocols in detail. Another good reference is *AppleTalk Network System Overview*, a technical description of the AppleTalk network system. Both books are written and produced by Apple Computer and published by Addison-Wesley Publishing Company. You can obtain these books at your local bookstore and from APDA.

APDA

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most APDA products. APDA offers convenient payment and shipping options including site licensing.

To order a product or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319
800-282-2732 (United States)
800-637-0029 (Canada)
716-871-6555 (International)
Fax: 716-871-6511

AppleLink:	APDA
America Online:	APDA
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you plan to develop Apple-compatible hardware or software products for sale through retail channels, you can get valuable support from the Developer Support Center by calling 408-974-4897 or using AppleLink (DEVSUPPORT).

Developers outside the United States and Canada should contact their local Apple office or distributor for information on local developer programs.

1 Overview of Client/Server Applications for the Network Server

The Network Server allows you to develop applications that integrate the power and services of AIX with the easy-to-use interface of the Macintosh interface. The applications you develop consist of components for both the AIX and Macintosh environments. They communicate over a network to form a single service.

The server component of this model runs on a Network Server and the Macintosh client component run on any Mac OS system. Macintosh clients can contact the Network Server for information and can perform tasks, such as file system management, on the server.

The two components of these client/server applications exchange messages using AppleTalk protocols and Apple events. Apple events provide a flexible, robust method for communication and relieve you of creating your own communication protocol.

This chapter begins by describing some sample client/server applications available for AIX. It then gives overviews of:

- the components of client/server applications
- the process by which client/server applications work
- the process you use to develop a client/server applications
- the protocols you use with a client/server application

Client/Server applications available on the Network Server

The following client/server applications are provided with AIX for the Network Server:

- Status Demo AppleTalk Services, a sample application for developers (see Chapter 4, “A Sample Application”)
- Fractal Demo AppleTalk Services, a sample application for developers (see Chapter 4, “A Sample Application”)
- Disk Management Utility, a Logical Volume Manager application for AIX for the Network Server
- CommandShell, a multi-windowed VT100-compatible terminal emulator

You can develop applications to perform other services. Examples of client/server applications for the Network Server that might be useful include

- Macintosh clients for Network Server applications, such as relational databases
- Modules that plug into an OPI publishing prepress server (OPI is an extensible pre-press interface that allows clients to request document manipulation operations on the server)
- remote print spooler management tools
- Network Server system monitoring applications
- InfoExplorer clients that allow Macintosh users to read the online documentation for the Network Server

Components of client/server applications

When you develop a client/server application, you create two separate programs that work together—a component for the server side that runs on the Network Server and a component for the client side that runs under the Mac OS. By developing each component separately, you can use the best tools for each environment. AIX for the Network Server provides several features that allow you to develop client/server applications.

The Network Server component

The server component of a client/server application consists of the program you create and uses several server features available on AIX for the Network Server. Here are the primary components involved in the server side of a client/server application:

- The Network Server component.

  This is the program you create that runs on the Network Server. The server component is a noninteractive UNIX daemon application.
- An AIX library implementation of the Apple Event Manager.

  This library is used by the Network Server component. This follows the Macintosh Apple Event Manager interface with a couple of additional routines.
- The PPC daemon (`ppcd`).

  The PPC daemon uses a configuration file (`/etc/ppcd.conf`) to get information about the Macintosh services available on the Network Server and advertise those services to clients. It also authenticates client requests and starts the Network Server component.

The Network Server component starts when the Macintosh client component of your client/server application contacts the PPC daemon on the Network Server.

The Network Server component and the PPC daemon communicate with the Macintosh client component using the AppleTalk Data Stream Protocol (ADSP) services through the Network Server AppleTalk stack. The PPC daemon also uses the NBP (Name Binding Protocol) as specified by the Macintosh Program-to-Program Communications protocol.

The Macintosh client component

The client side of a client/server application involves these features:

- The Macintosh client component—the program you create to provide the interface for the client/server application and to create a network connection. (The client uses Macintosh System 7 features to create the network connection.)

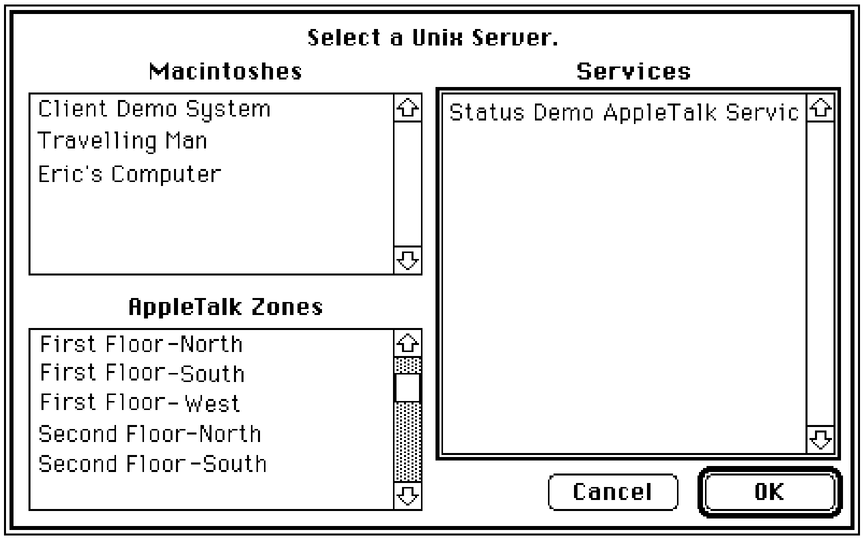
- The Network Server Passwd Tool. You need to install this extension when you install the Macintosh components from the *Mac OS Utilities for the Network Server With AIX* installation disk. For information about the installation process, see *Using AIX, AppleTalk Services, and Mac OS Utilities on the Network Server*.

How client/server applications work

To use client/server applications for the Network Server , users must establish communication between the Macintosh client and the Network Server.

To start a client/server application, the user simply double-clicks the application icon in the Finder on the remote Macintosh. The PPC Browser presents a dialog box to allow the user to select an AppleTalk zone, Network Server, and client/server daemon on that server.

Note: In this dialog box, the Macintoshes list corresponds to the Network Server and the Servers list corresponds to the particular AIX daemon.

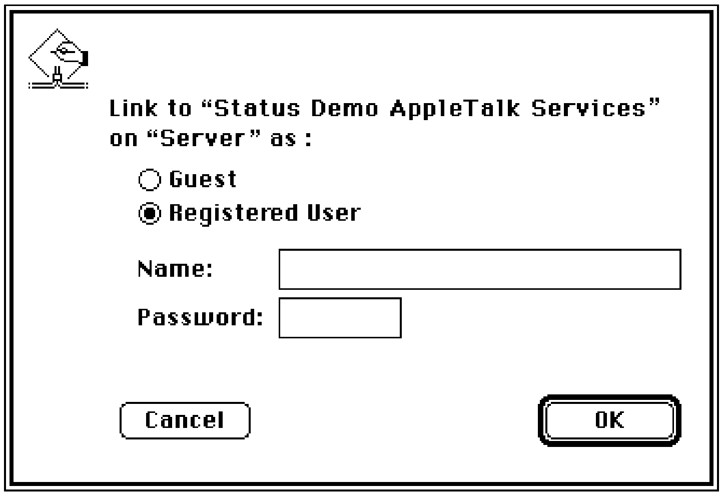


When a user selects a zone in the PPC Browser dialog box, the PPC Browser sends out a request to find all Network Servers in that zone. The PPC daemon on each Network Server in the zone responds to this request. When the user chooses a particular server, the PPC Browser requests a list of available services from the PPC daemon on that server.

The PPC daemon returns a list of client/server application services and some of their features—for example, information about the type of service and availability of guest access. This list of services appears in the PPC Browser dialog box.

A Macintosh client component can filter the list of available client/server application services so that the PPC Browser dialog box shows only the services related to the client component. For example, when it calls the PPC Browser, the Macintosh client can specify which types of services on Network Servers to display. The types are specified in the `ppcd.conf` file on the Network Server and provided by the PPC daemon when the Macintosh client queries the network for available servers.

After a user selects a service, the client displays the authentication dialog box for the user's name and password. The user must enter the name and password of a valid AIX user account on the Network Server.


Link to "Status Demo AppleTalk Services" on "Server" as :

○ Guest
[✓ ◉ Registered User]

Name:	
Password:	

The user name may be either a Network Server user name or the user’s full name as it appears in the password file on the Network Server. The password is encoded for transmission over the network.

Note: Some services allow guest access. This information is defined by the Network Server component of the client/server application and is sent with the list of services.

After the user has provided a name and password, the client sends the information to the PPC daemon on the server. The server verifies the accuracy of the user name and password. (For guest access, no password checking occurs.)

Once the user name and password has been verified, the PPC daemon starts the Network Server component of the client/server application and sets up a network connection between the Macintosh component and the Network Server component. Each connection from a Macintosh component gets a new Network Server process.

Before the PPC daemon starts the Network Server component, the PPC daemon uses the client’s user name to perform additional checks for security and determine access privileges. To control the level of access clients have, you can add a list of privileged users to an applications description in the `ppcd.conf` file. If a user is not listed in this file as a privileged user, your application’s server component could make a `setuid` system call to change the ID of the process to the user’s normal level. (The server component initially starts at `root` privileges.) Your application can also use the `ppcd.conf` file to examine the privileged user list and display or disable features of a client/server application based on the username and if it appears in the list of privileged users.

The `ppcd.conf` file is located in the `/etc` directory. For complete information about defining privileged users in the `ppcd.conf` file, see “Working With Security and Authorization” in Chapter 3, “Developing the Network Server Component.”

After the connection between the components of your application has been established, the PPC daemon closes its connection to the Macintosh client, and resumes listening for new requests. The client continues to use its same connection to communicate with the Network Server component.

Once both the client and server components of your application are running, they can exchange information through Apple events.

If one side of the application is not going to be communicating with the other component, the component may send “heartbeat” Apple events to let the other side know that the component is still active. The sample applications (described in Chapter 4) use heartbeats to notify components when to quit. Otherwise, processes for components could run indefinitely.

Developing a client/server application for the Network Server

The two halves of a client/server application are two separate programs. This simplifies application development because you can use the best tools available for each environment.

The Macintosh client component is a standard Macintosh application that uses Apple events. It connects to the Network Server component just as it would connect to another Macintosh computer.

The Network Server component does not need to worry about the network connection. The PPC daemon on the Network Server receives the Macintosh client request and starts the Network Server component. The Network Server component simply needs to call a library routine to initialize the Apple event communication.

Here is an overview of the steps involved in developing a client/server application:

1 **Determine the functionality for the Macintosh client component and the Network Server component.**

Decide which tasks each component performs.

2 **Determine which Apple events are needed to exchange information and how the two components must communicate.**

Note: You may need to register new Apple events with Apple Computer.

3 **Write the Macintosh client component.**

To create the Macintosh application, you need to be familiar with Apple events and the Macintosh interface. You can use any Macintosh development tools and debuggers as long as they allow access to the PPC Browser and Apple events.

4 **Write the Network Server component.**

Your server component should use the Apple Event Manager library to communicate with the client and use standard socket calls to detect input or output on the ADSP socket. For information about socket calls, see the AIX documentation.

5 **Test both components of the application and their communication.**

Because client/server applications use Apple events and the PPC daemon to set up the connection, you don't need to know any special network programming techniques. Apple events are essentially used as a remote procedure call (RPC) mechanism.

Note: If you prefer, you can use ADSP or the ADSP connection established by the PPC daemon instead of Apple events.

The only special development tool you need to use is a Network Server debugger that can attach to a running process, such as `dbx`. You need to use this kind of debugger because you rely on the PPC daemon to start the server component, and you cannot start the server component from the command line.

Using protocols

AppleTalk is the underlying communication channel for client/server applications. With AppleTalk, you can register an application name on the network and make it visible to all other computers on the network. The following AppleTalk protocols can be used with client/server applications for the Network Server:

- Apple Event Interprocess Messaging Protocol (AEIMP)

You must use AEIMP for client/server applications. You do not need to know the specifics of the protocol; you simply need to use the routines described in *Inside Macintosh,* Volume VI. Both the Macintosh and Network Server components can use these routines without modifications. AEIMP differentiates a high-level event from an Apple event, and the Apple Event Manager handles the details. Generic (developer-defined) high-level events are not currently supported.

- ADSP (AppleTalk Data Stream Protocol)

  To develop client/server applications, you do not need to be familiar with ADSP, but you can use it instead of Apple events if you prefer.

  ADSP is a connection-oriented, reliable, sequenced network protocol. (It is similar to TCP, with some added features.) The kernel ADSP interface is difficult to use: because there aren't any direct high-level calls to access the protocol, you must use the `ioctl` system call to fill out parameter blocks and post them to the system where the call completes asynchronously through a SIGIO signal handler. The response is always sent to the process that created the ADSP connection, which is not necessarily the process that posted the call.

  For more information, see the AppleTalk API section of the Developer's Kit.

- Macintosh PPC (Program-to-Program Communications)

  This protocol lets the Macintosh client and the Network Server component communicate over the network. It uses ADSP as its transport mechanism. The PPC Toolbox is used by server applications to make their presence known to the host system (and network) and by client applications to browse for servers to connect with.

  To develop a client/server application, you need to know enough about PPC to connect the two halves of your application. Specifically, you need to be familiar with the PPC Browser described in *Inside Macintosh* , Volume VI.

- Name Binding Protocol (NBP)

  The PPC daemon uses this protocol as specified by the Macintosh Program-to-Program Communications protocol.

2 Developing the Macintosh Client Component

The Macintosh component is the client side of a client/server application for the Network Server. It provides the interface for the client/server application.

This chapter introduces the information you need to develop the Macintosh client component. It describes

- Macintosh development environments
- connecting to a Network Server through the PPC Browser
- authenticating the connection
- using Apple events
- maintaining a network connection through “heartbeat” Apple events
- debugging and troubleshooting

Macintosh development environment

The Macintosh user interface is a unique environment. The Macintosh User Interface Toolbox is a collection of graphics routines accessible to all software in the system. You can use this Toolbox to incorporate the Macintosh interface into the Macintosh client component of a client/server application.

You can use any Macintosh development environment, tools, and debuggers as long as they give you access to the PPC Browser and Apple events.

The most commonly used development environments are MacApp and MPW (Macintosh Programmer's Workshop). For information on these development environments, see the *Programmer's Guide to MacApp*; *Macintosh Programmer's Workshop Development Environment*, Volume 1; *Macintosh Programmer's Workshop Development Environment*, Volume 2; *MPW Pascal; Macintosh Programmer's Workshop C*. These books are available from APDA (see the Preface for ordering information).

The Macintosh client component of a client/server application also runs properly on Sun and Hewlett-Packard systems that are running the Macintosh Application Environment 2.0, which supports AppleTalk.

For complete information about developing Macintosh applications, see *Inside Macintosh.* For information about human interface guidelines, see *Macintosh Human Interface Guidelines*.

Using the PPC Browser to create a network connection

You use the PPC Browser to locate and connect to the Network Server component of your client/server application. The client component of the application needs to make a call to display the PPC Browser dialog box, which allows the user to select the desired AppleTalk zone, Network Server, and service. When the user selects a zone in the PPC Browser dialog box, the PPC Browser sends out a request to find all Network Servers in that zone. The PPC daemon on each Network Server in the zone responds to this request. When the user chooses a particular server, the PPC Browser requests a list of available services from the PPC daemon on that server.

The PPC daemon returns a list of Network Server services and some of their features—for example, information about the type of service and availability of guest access. This list of services appears in the PPC Browser dialog box.

The following routine displays the PPC Browser dialog box and searches for Network Server components.

```
err = PPCBrowser (prompt, applListLabel, defaultSpecified,
     theLocation, thePortInfo, portFilter,
     theLocNBPType);
```

If the `PPCBrowser` routine returns `noErr` after the user has selected a server and service, the parameters `theLocation` and `thePortInfo` specify the connection chosen by user. If the routine returns a `userCanceledErr` result code, the user clicked the Cancel button and no port was selected.

The parameters to the routine have the following values:

- `prompt`

  The `prompt` parameter is a string of text that the PPCBrowser routine displays as a prompt in the PPC Browser dialog box. If you specify `NIL` or pass an empty string, the default prompt "Choose a program to link to" is used.

- `applListLabel`

  The `applListLabel` parameter is a string that specifies the title of the list of PPC ports in the dialog box. If you specify `NIL` or pass an empty string, the default title "Programs" is used.

- `defaultSpecified`

  If you set this parameter to `true`, the `PPCBrowser` routine tries to highlight the PPC port specified by the parameters `theLocation` and `thePortInfo` when the PPC Browser dialog box first appears. If that port cannot be found, the routine highlights the first PPC port in the list.

- `portFilter`

  The `portFilter` parameter determines which ports appear in the PPC Browser dialog box. If you specify `NIL`, the names of all existing PPC ports are displayed; otherwise, you must specify a pointer to a port filter function. (For details, see the next section, "Filtering Servers in the PPC Browser.")

- `theLocNBPType`

  This parameter specifies the NBP type passed to `NBPLookup` to generate the list of servers. If you specify `NIL` or pass an empty string, the default `PPCToolBox` is used. Note that the current computer is always included in the list of servers.

The `PPCBrowser` routine fills in two data structures: a `LocationNameRec` structure and a `PortInfoRec` structure, neither of which, by itself, is a valid destination for an Apple event from the Macintosh component. Instead, you must create a third data structure of type `TargetID`, and fill it with all of the `LocationNameRec` information and part of the `PortInfoRec` information. You configure the target address for Apple events from the Macintosh component as an `AEAddressDesc` structure by setting its `descriptorType` field to `typeTargetID` and its handle to the contents of the `TargetID` structure as shown in this sample code.

```
AEAddressDesc       targetAddr;

LocationNameRec     theLocation;

PortInfoRec         thePortInfo;

TargetID       theTargetID;

osErr = PPC Browser("\pTitle", "\pPrograms", false, &theLocation,
     &thePortInfo, MyPPC BrowserFilter, "");

theTargetID.location = theLocation;

theTargetID.name     = thePortInfo.name;

targetAddr.descriptorType = typeTargetID;

PtrToHand((Ptr)&theTargetID, &(targetAddr.dataHandle),
     sizeof(theTargetID));
```

The target address is required for all future communication with the Network Server component.

For complete information about using the PPC Browser, see the PPC Toolbox chapter in *Inside Macintosh*, Volume VI.

Filtering Servers in the PPC Browser

A Macintosh client component can filter the list of Network Servers in the PPC Browser dialog box to show only servers that provide services related to the client component. When you call the PPC Browser, you can specify which types of servers to display. The types are previously specified in the `ppcd.conf` file on the Network Server side and provided by the PPC daemon when the Macintosh client queries the network for available servers. (For information about the `ppcd.conf` file and specifying application types, see "Using the PPC Daemon Configuration File" in Chapter 3, "Developing the Network Server Component.")

For example, you can choose to display only servers for the Status Demo AppleTalk Services application (of type `'JVLN'` in the `ppcd.conf` file).

To filter servers in the PPC Browser, you need to define a filter function and pass this function as a parameter to the `PPCBrowser` routine.

The following sample function illustrates how you use a filter. In this example, the `MyPPCBrowserFilter` function returns `true` for ports with the port type string `'JVLN'`.

```
static pascal Boolean MyPPCBrowserFilter(LocationNamePtr
/* theLocation */, PortInfoPtr thePortInfo)
{
     OSType    type;
     if (thePortInfo->name.portKindSelector ==
          ppcByString)
     {
          BlockMove(thePortInfo->name.u.portTypeStr + 1,
               Ptr(&type), sizeof(type));
          // The BlockMove is so that we don't get an
          // address error on a 68000-based machine
          // due to referencing a long at an odd-address.
          if (type == JVLN)
               return TRUE;
     }
     return FALSE;
}
```

The `PPCBrowser` routine calls your filter function once for each port on the selected server. Your function should return `TRUE` for each port you want to display in the PPC Browser dialog box and `FALSE` for each port that you do not want to display. Do not modify the data in the filter function parameters `theLocation` and `thePortInfo`.

Authenticating a network connection

Before the Macintosh client component of your application can connect to a server, you must run the AppleTalk Services installation program from the floppy disk titled *Mac OS Utilities for the Network Server*. (This disk is provided with AIX for the Network Server.) During the installation process, place the Network Server Passwd Tool in the System Folder on the Macintosh client. The Macintosh component uses this system extension to support encoded password communication and it is required for all client/server applications.

After a user has selected a Network Server component to connect to from the PPCBrowser, the PPC daemon on the server sends the Mac OS client a request to authenticate the user, and it displays the authentication dialog box for the user's name and password. You do not need to call a routine to provide the user authentication dialog box.

Note: Some services allow guest access. Guest access is defined by the `/etc/ppcd.conf` file and is sent to the client with the list of services. For guest access, the server does not require a password.

Using Apple events

To write the Macintosh client component of your application, you need to be familiar with Apple events. After your Macintosh component has established a connection with the Network Server component and has obtained the Network Server's address from the PPC Browser, the components can exchange information using Apple events.

IMPORTANT The Network Server component of a client/server application does not start running until an Apple event is sent from the Macintosh component.

Apple events from the Macintosh component may contain commands to be run on the Network Server, such as `ls`, `date`, or `format`. Apple events from the server component may contain the results of the commands.

You can use any of the Apple event routines described in the Apple Event Manager chapter of *Inside Macintosh*, Volume VI. For most clients, you only need to use a few Apple Event Manager routines. You can use commands to process, create, and send information. For some Apple events, you can also specify that you want a reply from the server.

Follow these steps to send information to the server (the routines are fully detailed in *Inside Macintosh*):

1. **Create an Apple event with the `AECreateAppleEvent` routine.**
2. **Include the necessary information in the Apple event with the `AEPutParamPtr` routine.**
3. **Send the Apple event with the `AESend` routine. Send the event to the address received from the PPC Browser.**

When you receive an Apple event from the Network Server component, use the `AEGetParamPtr` routine to extract information from it.

Maintaining a network connection

When the Macintosh client component is connected to the Network Server component, it can periodically check to ensure that the server component is still active. It does this with heartbeat Apple events. The server component sends heartbeat Apple events to ensure that the network connection is working and the client has not shut down. The client component can check to see if the server component has sent heartbeats and inform the user if heartbeats have not been received.

To use heartbeat Apple events, you need to keep track of when you received the last Apple event. After the Macintosh component receives a heartbeat Apple events or other Apple event, it returns its heartbeat counter to 0. If one component has not received a heartbeat or an Apple event in a specified period of time, the component shuts down.

This section provides sample code for checking heartbeats. (Complete sample code is available in `/usr/lpp/apple.remoteutils/src/client`.)

The following are variables used in determining when to send heartbeats and whether to shut down. The variables keep track of when the last heartbeat was received and when the component last checked for a heartbeat. Another variable keeps track of how many times the component has checked for a heartbeat without find finding one. When this value reaches a predetermined number (3 in the sample code), the component assumes the other component has quit and it shuts down. At first, these variables are set to 0.

```
long time_of_last_recv=0;
long time_of_last_check=0;
int missed_heartbeats=0;
Boolean TimeToQuit=false;
```

The following code checks to see if a heartbeat Apple event has been received since the last check. If you reach the maximum number of checks for heartbeats without a response from the server component, the Macintosh component should quit.

```
void chk_heartbeat(void)
{
    time_of_last_check = TickCount();
    if ((time_of_last_check - time_of_last_recv) >
HEARTBEAT_CHECK) {
        missed_heartbeats++;
        if (missed_heartbeats >= MAX_MISSED_HEARTBEATS) {
            TimeToQuit=TRUE;
        }
    }
    else
        missed_heartbeats = 0;
}
```

When the Macintosh client component quits, it must inform the Network Server component that the connection is closing down so that the server component can take appropriate action before quitting. If the Macintosh component were to quit without informing the Network Server component, the Network Server process for the application could run indefinitely.

Debugging

To debug a client/server application, you can any Macintosh debugging tool that suits your application. Some of the available debuggers include:

- SourceBug
- MacsBug,
- Jasiks debugger

3 Developing the Network Server Component

The Network Server component of a client/server application is a noninteractive AIX daemon application, executing on the Network Server. After the Macintosh client component makes a connection, the PPC daemon starts the Network Server component. The Network Server component executes commands for the Macintosh component, which is controlled by the user.

Users cannot start the Network Server component from the command line or initiate a connection with the Macintosh component from the Network Server.

This chapter details how to develop the Network Server component of a client/server application. It discusses

- the Network Server development environment
- the header files and libraries provided with AIX for the Network Server
- the network connection between the components
- Apple events used by the Network Server component
- creating the main program loop for the Network Server component
- security issues
- the configuration file for the PPC daemon
- debugging issues

Network Server development environments

To develop the Network Server component, you can use any UNIX C compiler and development environment. You must use a debugger, such as `dbx`, that can attach to a running process.

Header files and libraries

AIX for the Network Server provides several header files and libraries to help you develop client/server applications for the Network Server.

The following header files are available in `/usr/include/mac/*.h`:

- `AppleEvents.h`

  This header file contains standard Apple event definitions.

- `AUXAESuite.h`

  This header file contains Apple event definitions specific to the applications provided by Apple Computer.

- `Types.h`

  This header file contains type definitions.

The following shared libraries are available in `/usr/lib`:

- `libaem.a`

  This library holds all the Apple Event Manager routines.

- `libat.a`

  This library contains all the AppleTalk routines for the Network Server component.

- `libadsp.a`

  This library contains all the ADSP Manager routines you need if you choose to use ADSP instead of Apple events.

Creating and maintaining a network connection

The Network Server component cannot establish the network connection; the Macintosh client component contacts the PPC daemon on the Network Server and the PPC daemon authenticates the Macintosh request. When the PPC daemon on the Network Server receives the Macintosh request and authenticates it, the daemon starts the appropriate Network Server component for the application. Once started, the Network Server component needs to call a library routine to initialize the Apple event communication.

Note: The Network Server component is not started until the Mac OS client sends an initial Apple event intended for the Network Server component.

To ensure that one component does not continue running after the other side has shut down, the components of a client/server application can keep track of each other. If Apple events are not exchanged regularly, the server component can send a "heartbeat" Apple event to the client to ensure that the client is still active and has not quit. This is important because if the Macintosh component were to quit without notification, the Network Server process could run indefinitely. Heartbeats ensure that components of client/server applications do not run indefinitely after the other side of the application has quit. It is possible to clean up components of client/server applications other ways after the client side has unexpectedly quit, but we recommend using heartbeat Apple events.

To use heartbeats, the server component sends a heartbeat Apple event to the client. If the WaitNextAppleEvent routine returns a value of `-1`, the network connection has been broken and the server component shuts down.

You need to keep track of when you sent the last Apple event. After the server receives a heartbeat Apple event or other Apple event, it returns its heartbeat counter to 0. If one component has not received a heartbeat or an Apple event in a specified period of time, the component shuts down.

You can use the `snd_heartbeat` command (in `misc.c`) to send heartbeats to the other side.

This section provides sample code for working with heartbeat commands. (Complete sample code is available in the `/usr/lpp/apple.remoteutils/src/server` directory. See Chapter 4 for complete details.)

The following are variables used in determining when to send heartbeats and whether to shut down. The variables keep track of when the last heartbeat was sent and when the values where last updated. At first, these variables are set to 0.

```
    time_t time_of_last_send = 0;

    time_t time_of_last_update = 0;
```

The next command sets the current time. You use the time for later calculations.

```
   (void)time(&time_now);
```

The next line of code determines whether it is time to send another heartbeat command to the client. If the difference between the current time and the time the last heartbeat was sent is greater that the `HEARTBEAT_SEND` value, the `snd_heartbeat` command sends a heartbeat to the client.

```
   if (difftime(time_now, time_of_last_send ) >= HEARTBEAT_SEND
)    snd_heartbeat();
```

Using Apple events

The Network Server component uses Apple events to register event handlers and to respond to incoming events and provide replies as requested.

When the Network Server component starts, it needs to call one library routine to initialize the Apple Event Manager:

```
AEInit ()
```

After you make this call, your application can exchange Apple events with the Macintosh client component.

You can use any of the Apple event routines described in the Apple Event Manager chapter of *Inside Macintosh*, Volume VI. Most client/server applications need to use only a few Apple event routines. The routines most likely to be useful are `AEPutParamPtr`, for placing information in an Apple event, `AEGetParamPtr`, for extracting information from an Apple event, and `AESend`, for sending the Apple event.

The only Apple event routine that behaves differently from its Macintosh implementation is `WaitNextAppleEvent` which the Network Server component uses to retrieve Apple events from the Macintosh component. This routine is similar to the Macintosh Toolbox routine `WaitNextEvent`. The main difference is that only events of type `kHighLevelEvent` will be returned, so there is no need to check the event type before you send it to `AEProcessAppleEvent`.

Additionally, the time-out value for the `kHighLevelEvent` routine is different from that used in the Macintosh implementation. In the Network Server implementation, a AIX `timeval` structure lets you specify a time in seconds and microseconds. The time-out parameter can be used in three different ways:

- If you specify a filled-in structure that determines how long `WaitNextAppleEvent` waits for an event, and if an event arrives before the time-out interval expires, `WaitNextAppleEvent` returns and does not wait for the rest of the interval.
- If you specify 0 for the time-out value, `WaitNextAppleEvent` checks for an event. If there is an event to process, the routine returns a value of `true` and fills in the event record. If an event is not available, the routine returns `false`.
- If you pass `NULL` in place of a `timeval` structure, `WaitNextAppleEvent` waits for an event and blocks the calling Network Server process.

All the other Apple Event Manager routines can be used as described in *Inside Macintosh*, Volume VI, and all AIX libraries and system calls can be used as they are described in the AIX documentation.

Writing the main program loop

The main program of the server component may need to monitor input from several processes at one time. You might need to switch control between waiting for and processing Apple events and performing client tasks on the server. To do, create a loop that follows this format:

```
Until instructed to quit:
Wait for Apple events;
If an Apple event contains a task for the server;
    Process Apple event by matching it with a routine listed in the Apple
    event handler table;
    end
end
```

Use the `select` system call to wait for input from a process on the server and to wait for Apple events at that same time.

The `select` system call lets you maintain your connection with the client while you are performing tasks that require some time. For example, if the client component instructs the server component to format a hard disk, the command may take some time. While the command is executing, the server side needs to wait for the outcome, but it still needs to listen to the client for additional commands and heartbeats.

To use the `select` call, refer to the `select` manual page.

To see an example of a main program loop, see the `/usr/lpp/apple.remoteutils/src/server/javelin.c` file. See Chapter 4, “A Sample Application,” for more information about the sample files.

Setting up and starting the PPC daemon

The PPC daemon supports client/server applications and manages the exchange of AppleTalk communications between Macintosh client components and the Network Server.

You can start, stop, and customize the PPC daemon through the Program-to-Program Communication option in the SMIT AppleTalk Services menu. You can also use this option to add new daemons to the server that you have created for Macintosh clients. For complete information about using the PPC daemon with SMIT, see *Using AIX, AppleTalk Services, and Mac OS Utilities on the Network Server*.

To use client/server applications, you must start the PPC daemon. To start the PPC daemon:

1. **Click "Start the 'ppcd' daemon."**

   This option appears in the Program-to-Program Communication menu available from the AppleTalk Services menu.

   A dialog box appears.

2. **Specify when you want to start the daemon.**

   You can start the daemon now, at the next system restart, or both.

3. **Provide a hostname for you computer.**

4. **Click OK.**

   The daemon starts.

In addition to starting the PPC daemon, the PPC daemon selection under the AppleTalk Services item contains the following options:

- Stop the 'ppcd' Daemon
- List all 'ppcd' Services

  This option displays all the daemons on the server used for client/server applications such as the Disk Management Utility, the Status Demo AppleTalk Services application, and CommandShell. If you have added any additional daemons, they will also be displayed.
- Add a 'ppcd' Service

  This option lets you add a new daemon for a client/server application to your system. When you choose this option, a dialog box appears that lets you specify information for the daemon.

To add a PPCD service, you need to specify:

- Name of the service. Specify the name of the PPC component as you want it to appear in the PPCBrowser dialog box for the Mac OS client. Do not use spaces in the name.
- User ID. Provide a user name or user ID. This value determines the privileges the application has when it runs.
- Group ID. Provide a group name or group ID. This value determines the privileges the application has when it runs.
- Path of the daemon executable. Provide the full path to the server component (daemon).
- Daemon signature. The signature is a unique four-letter code that identifies the daemon. Mac OS clients use the signature to filter client/server applications and search for specific types of applications.
- Guest access. The guest field is optional. If you select guest access, any client can connect to the application as the guest user (an actual account). If you choose not to have guest access, the user must supply a valid account (user name) and password to start the client/server application.
- Privileged users. If you want specific users to be able the run an application with extra privileges (such as root privileges), provide their names in this field. If a user is listed in this field, the user ID and group ID values are not used.

- Change/Show Characteristics of a ‘ppcd’ Service

  After you select this option, select a service (daemon) to examine. The same options appear as those for the Add a ‘ppcd’ Service option. You can change any of them or just examine them.

- Remove a ‘ppcd’ Daemon

  Deletes an entry for a daemon from the PPC daemon configuration file.

The PPC daemon uses the `ppcd.conf` file as the basis for the information it advertises to clients. The `ppcd.conf` file can also be used to provide guest access. The `ppcd.conf` file is located in the `/etc` directory.

Each client/server application needs an entry in the configuration file, `/etc/ppcd.conf`, in order for the PPC daemon to be able to start the Network Server component.

Note: When you add a new client/server application, you need to send the PPC daemon process a “hangup” signal to have it reread its `ppcd.conf` file or you should stop and restart the PPC daemon.

Working with security and authorization

When the PPC daemon starts the Network Server component, the component runs with `root` privileges. You can use the `setuid` system call in your application to change permissions and modify the way specific users can access the information on the server and run commands.

Before the PPC daemon starts the Network Server component, it examines the user’s name. The PPC daemon can use this name to determine the level of client access before it starts the Network Server component. The PPC daemon uses the last field of the `ppcd.conf` file to determine if the specified user has special privileges. The PPC daemon can check the `ppcd.conf` file to see if the specified user has special privileges. If the user name is in the `ppcd.conf` file , the PPC daemon can let the user maintain full privileges. If the user name is not in the `ppcd.conf` file , the PPC daemon can call the `setuid` system call and change the permissions of the process to the user’s normal access permissions.

Debugging and troubleshooting

The only special development tool you must use is a UNIX debugger that can attach to a running process, such as the `dbx` debugger. This is necessary because the server component must be launched by the PPC daemon, which does the connection setup.

If your Network Server component quits before you can attach to the running process, or if you have problems with the startup of the Network Server component, you can instruct the daemon to sleep for two minutes before doing any initialization. The two minutes of inactivity will give you enough time to attach to the process. Be sure to insert the `sleep` command before your application performs any tasks.

To instruct the Network Server component to sleep, add the following code to your application before the Network Server component initializes any values and performs any tasks.

```
#ifdef SLEEP
    #ifdef DEBUG
        fprintf(debugfp,"STATUS DEMO APPLETALK SERVICES:
Sleeping 120 seconds to allow dbx to attach...\n");
    #endif
    sleep(120);
#endif
```

4 A Sample Application

This chapter discusses the files used by Status Demo AppleTalk Services, a sample client/server application that obtains system information from a Network Server and displays it on a remote Macintosh. The Status Demo application is provided with AIX for the Network Server, and you can find its complete source code in the `/usr/lpp/apple.remoteutils/src` directory. In this directory are two subdirectories, `/client` and `/server`. These directories separate the code for the Macintosh client component and the Network Server component.

The code shows how Apple events are exchanged, how the network connection is established, how the connection is kept active, and how the Network Server component collects and sends system information.

For the client side, there are two examples of the code. They both work with the server component. One is in the CodeWarrior developer environment and uses Metrowerks (C). The other example is in the Macintosh Developer's Workshop (MPW) and uses the MacApp 3.0.1 framework.

The Macintosh client component

You can use many different compilers and approaches for the Macintosh client component of a client/server application. Two types of sample code are provided for the Status Demo client component: procedural (C in Metrowerks CodeWarrior) and object-oriented (C++ in MPW/MacApp).

To develop the Macintosh component, you need to include code to communicate with the Network Server component, handle Apple events, and provide the Macintosh interface.

The Metrowerks C version of the Status Demo client component uses the following files:

- `javelin.c`

  This file contains the main program loop of the Status Demo application and performs such tasks as displaying the PPC Browser, filtering available services in the PPC Browser, and handling Apple events.

- `symbiont.c`

  The calls in this file can be used in any client/server application. They perform such tasks as sending and checking heartbeat Apple events, defining Apple event handlers, and exchanging version information. This file also contains generic Macintosh routines for mouse and window operations.

- `javelin.h`

  This file is the header file for information specific to the Status Demo application.

- `JavelinEvents.h`

  This file defines Apple events for the Status Demo application.

- `symbionts.h`

  This file defines heartbeat values, such as the intervals at which to send and check for heartbeat Apple events and the number of missed heartbeats your component detects before shutting down. This file also defines other constants for the application.

- `version.h`

  This file contains the version number of the application. The version number is exchanged between an application’s server and client components to ensure that they can exchange information properly.

The MPW/MacApp version of the Status Demo client component uses the following files:

- `Javelin.MAMake`

  The make file for the application.
- `Javelin.r`

  This file contains the resources.
- `JavelinIcePick.rsrc`

  This file contains the view resources.
- `MJavelin.cp`

  This file contains main program.
- `UJavelin.cp`

  This file contains the application classes.
- `UClientCommands.cp` and `UClientCommands.h`

  These files manage received Apple events.
- `UCommandJavelin.cp` and `UCommandJavelin.h`

  These files call the PPC Browser.
- `UDocumentJavelin.cp` and `UDocumentJavelin.h`

  These files contain the document classes.
- `UJavelinAppleEvent.cp` and `UJavelinAppleEvent.h`

  These files contain the routines for Apple events.
- `UJavelinDialogs.cp` and `UJavelinDialogs.h`

  These files contain the dialog boxes and routines for error conditions.
- `UJavelinView.cp` and `UJavelinView.h`

  These files contain the view routines.
- `URemoteCommands.cp` and `URemoteCommands.h`

These files contain routines to send Apple events.

- `AUXSuite.h`

  This file contains descriptions of Apple events.

The Network Server component

This section describes the files used by the Network Server component of the Status Demo application.

- `javelin.c`

  This is the primary file for the Status Demo application. This file initializes the Network Server to use Apple events, sets up a `select` loop to listen for input from various sockets, and maintains the connection with the client component by sending heartbeat Apple events. The file also determines how to respond to incoming Apple events.

- `handler.c`

  This file contains the Apple event handlers for the Status Demo Network Server component.

- `misc.c`

  This file contains miscellaneous routines for the Status Demo program, including those that check and send heartbeats and those that get and send information in the form of Apple events.

- `javelin.h`

  This file is the header file for information specific to the Status Demo application.

- `misc.h`

  This file defines heartbeat values, such as the interval at which to send and check for heartbeat commands and the number of missed heartbeats your component detects before shutting down. This file also defines other constants for the application.

- `version.h`

This file contains the version number of the application. The version number is exchanged between an application's server and client components to ensure that they can exchange information properly.

- `AEregistry.h`

  This file contains C interfaces to Apple Event Registry.

- `AppleEvent.h`

  This file contains C interfaces to Macintosh libraries for handling Apple events.

- `AUXSuite.h`

  This file defines Apple events for the Status Demo Network Server component.

- `mac/types.h`

  This file contains data structures and type definitions for Apple events and Macintosh routines.

- `UEPPC.h`

  This file contains C interfaces to the Macintosh libraries for the PPC Browser and PPC Toolbox.

- `UPPCToolbox.h`

  This file contains C interfaces to the PPC Toolbox.

Section II Developing Device Drivers

This section supplements the AIX Version 4.1 book *Writing a Device Driver*. You should be familiar with the material in that guide before you read this one.

You should have a working knowledge of the C programming language, and you should have some experience writing device drivers. You should also be familiar with the AIX operating system. If you need to learn more about AIX, see the section “For More Information” at the end of this introduction.

This guide provides information about the modifications made to AIX so that it runs on the Network Server. The chapters are organized as shown here:

- Chapter 5, “Overview of Changes With the Network Server,” describes hardware and software differences that relate to all aspects of device driver development.
- Chapter 6, “The Open Firmware Device Tree,” describes the Network Server system startup, device discovery, and the device tree that stores values for devices.
- Chapter 7, “Device Configuration on the Network Server,” details the new device hierarchy, changes to the Object Database Manager (ODM), and configuration methods.
- Chapter 8, “Device I/O on the Network Server,” describes changes to the I/O subsystem and Direct Memory Access (DMA).
- Chapter 9, “The Network Server Interrupt Subsystem,” highlights differences in interrupt levels and mapping.
- Chapter 10, “Implementing Graphic Input and 2D Graphic Device Drivers,” highlights the differences between IBM’s and Apple’s graphics libraries.

Also, refer to the following chapter for additional information.

- Appendix, “Keyboard Positions,” contains information about the keyboard for the Network Server and describes differences in key positions for international keyboards..

How to use this guide

This guide is a supplement to the AIX 4.1 book *Writing a Device Driver*. Use that book for an overview of device driver development, general information, definition of terms, and general procedures. After you are familiar with the general device driver development, refer to this guide to learn about the specific changes to device drivers for AIX on the Network Server.

For more information

The following technical books, documents, and resources provide additional information about developing device drivers.

Most important, you'll want to keep this book on hand for reference:

- AIX Version 4.1 *Writing a Device Driver.* Available from IBM, order number SC23-2593.

Other documentation that you may find useful:

- AIX Version 4.1 *Commands Reference.* Available from IBM, order number SBOF-1851.
- AIX Version 4.1 *General Programming Concepts: Writing and Debugging Programs.* Available from IBM, order number SC23-2533.
- AIX Version 4.1 *Communications Programming Concepts.* Available from IBM, order number SC23-2610.
- AIX Version 4.1 *Kernel Extensions and Device Support Programming Concepts.* Available from IBM, order number SC23-2611.
- AIX Version 4.1 *Files Reference.* Available from IBM, order number SC23-2512.
- AIX Version 4.1 *Problem Solving Guide and Reference.* Available from IBM, order number SC23-2606.
- AIX Version 4.1 *Technical Reference, Volume 5: Kernel and Subsystems.* Available from IBM, order number SC23-2618.
- AIX Version 4.1 *Technical Reference, Volume 6: Kernel and Subsystems.* Available from IBM, order number SC23-2619.
- *UNIX System V Release 4, Programmer's Guide: STREAMS*. Englewood Cliffs, N.J.: Prentice-Hall, 1990.
- Angebranndt, Susan, Raymond Drewry, Philip Karlton, Todd Newman, Keith Packard, and Robert W. Scheifler. *Strategies for Porting the X v11 Sample Server*. Massachusetts Institute of Technology,1991.
- Fortune, Erik and Elias Israel. *The X Window Server*. Digital Press.

- Gettys, James, Ron Newman, and Robert W. Scheifler. *Xlib—C Language X Interface*: *MIT X Consortium Standard, X Version 11, Release 5*. MIT X Consortium, Massachusetts Institute of Technology, 1991.
- Patrick, Mark, and George Sachs. *X11 Input Extension Library Specification: MIT X Consortium Standard, X Version 11, Release 5.* Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology, 1989, 1990, 1991.
- Patrick, Mark, and George Sachs. *X11 Input Extension Protocol Specification: MIT X Consortium Standard, X Version 11, Release 5.* Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology, 1989, 1990, 1991.
- Sachs, George. *X11 Input Extension Porting Document. MIT X Consortium Standard*. X Version 11, Release 5 . Hewlett-Packard Company and the Massachusetts Institute of Technology. 1989, 1990, 1991.
- Scheifler, Robert W. *X Window System Protocol: MIT X Consortium Standard, X Version 11, Release 5.* MIT X Consortium, 1991.

APDA

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most APDA products. APDA offers convenient payment and shipping options including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog,* contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319
800-282-2732 (United States)
800-637-0029 (Canada)
716-871-6555 (International)
Fax: 716-871-6511

AppleLink:	APDA
America Online:	APDA
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you plan to develop Apple-compatible hardware or software products for sale through retail channels, you can get valuable support from the Developer Support Center by calling 408-974-4897 or using AppleLink (DEVSUPPORT).

Developers outside the United States and Canada should contact their local Apple office or distributor for information on local developer programs.

5 Overview of Changes for the Network Server

This guide details the hardware and software differences you may encounter when you develop a device driver for the AIX operating system for the Network Server as opposed to AIX Version 4.1 for other platforms. This guide does not replace IBM documentation for device drivers, rather, it highlights modifications and supplements the IBM documentation.

The majority of changes to AIX for the Network Server are due to the differences in firmware and hardware between the Network Server and the IBM AIX platforms. Changes have been made in the AIX 4.1 subsystem to reflect differences in hardware and to use the Network Server's Open Firmware functionality for device information. AIX for the Network Server makes most of these differences invisible to the user and maintains the same interfaces, routines, and structure.

This chapter introduces the differences in AIX for the Network Server for developers of device drivers. The following chapters provide more detail about the changes.

Hardware differences

The operating system for the Network Server consists of the AIX 4.1 operating system ported to the Network Server hardware platform. Although the operating system remains essentially the same, the underlying hardware is different.

The hardware differences do not effect the user and seldom affect the developer. Wherever possible, the same hardware interface has been maintained.

The Network Server uses the PCI bus architecture (instead of another bus architecture, such as MicroChannel). The Network Server also uses Open Firmware to configure devices during startup. Chapter 6, “The Open Firmware Device Tree,” describes the new startup process.

The Network Server device hierarchy is also different from other AIX device hierarchies. Chapter 7, “Device Configuration,” outlines these changes.

The Network Server’s interrupt hardware is considerably different from that of the IBM RS/6000. The interrupt interfaces are the same, although the AIX interrupt subsystem has been modified to account for the hardware differences.

Despite these hardware differences, the AIX operating system is essentially preserved on the Network Server. Several software differences result from the hardware differences, but most of these are hidden from the user.

Software differences

If you are a developer of device drivers and are already familiar with AIX 4.1, you need to be concerned with the following differences in the software:

- Open Firmware

  The Open Firmware startup process changes the way devices are configured and changes the way you access certain information, such as addresses and interrupt levels.

- device I/O

  The Network Server has a different on-board I/O controller.

- DMA

  DMA (direct memory access) is different, and DMA support services are not used. The IBM platform shared DMA through a DMA engine. With the Network Server, each device has an individual DMA engine.

- device configuration methods

  With the new device hierarchy for the Network Server, device configuration methods use Open Firmware. Open Firmware provides new routines for accessing information created during the Open Firmware startup process. The content of the ODM databases reflects the Network Server hardware.

- graphics device drivers

  3D features are not supported

- network device drivers use the `m_nextpkt` field

  AIX for the Network Server uses the `m_nextpkt` field of the `mbuf` structure to pass multiple packets to network drivers. Thus, the `mbuf` chain passed to the output routine of your driver will be two-dimensional: the `m_next` field from the first `mbuf` pointer links the fragments of the first packet; the `m_nextpkt` field of the first `mbuf` pointer will (if non-NULL) point to the second packet.

Note: IMB's `mbuf` structure allows for this; however, their protocol stacks currently do not take advantage of this feature, and their network drivers typically do not follow the `m_nextpkt` pointer. Currently, the AIX DLPI component does not pass packet chains down to network drivers, except for the specific case of the AppleTalk stack.

The following areas of device driver development have <u>not</u> been changed from IBM's AIX:

- synchronization and serialization
- block device drivers
- writing a virtual file system
- stream-based TTY interface

If you need information about these topics, see the IBM documentation. (*Note:* The IBM documentation may refer to hardware that does not apply to the Network Server.)

6 The Open Firmware Device Tree

The Network Server uses Open Firmware to configure the devices and start the operating system.

When the Network Server starts up, Open Firmware searches for devices and builds a data structure of nodes called a *device tree,* in which each device is defined. When Open Firmware has discovered devices and built the device tree, it starts the AIX operating system. AIX for the Network Server uses the device tree to learn about available devices and build Object Database Manager (ODM) device databases. To get device information from the device tree, your configuration methods use Open Firmware routines.

The Open Firmware startup process conforms to *IEEE Standard 1275* and the *PCI Bus Binding to IEEE 1275-1994* specification.

This chapter provides an overview of the startup and device configuration process, including

- system startup and device discovery
- the Open Firmware device tree
- obtaining device information from the device tree
- how the device tree relates to the ODM databases
- the Open Firmware routines and command-line interface

Chapter 7, “Device Configuration for the Network Server,” provides more information about the device hierarchy and device configuration methods.

System startup and device discovery

The Open Firmware startup process is driven by startup firmware (also called *boot firmware*) in the Network Server ROM and in memory chips on PCI cards (expansion ROM). During the Open Firmware startup process, startup firmware in the Network Server's ROM searches the PCI buses and generates a data structure, the device tree, that lists available PCI devices. The device tree also lists the support software, including drivers, provided by each expansion card. The startup firmware then finds the operating system on a device and loads it.

The Network Server firmware provides capabilities to query the hardware about device characteristics and build a device tree that an operating system can access to get information about the hardware. The operating system uses Open Firmware routines to access the information in the device tree.

About the device tree

The Open Firmware device tree is a data structure that describes the set of devices and services available to the system. The user and the operating system can use the device tree to examine the system's hardware configuration.

Devices are attached to a Network Server on a set of interconnected buses. Open Firmware represents the buses and the attached devices as nodes on the device tree. Open Firmware constructs the initial device tree based on built-in devices; the device tree grows as new devices are discovered during startup.

For example, the device tree for the Network Server, which uses the PCI bus, starts with a node for each PCI bridge. Open Firmware descends each node to look for other devices attached to it. This process is recursive, resulting in a hierarchical device tree.

Listing 6-1 shows a device tree for the Network Server. In this listing, child devices are indented under their parent devices.

```
/PowerPC,604@0
     /12-cache@0,0
/memory@0
/AAPL,ROM@FFC00000
/bandit@F20000000
     /53c825@11
     /53c825@12
/gc@10
     /53c94@1000
     /mace@11000
     /escc@13000
          /ch-a@13020
          /ch-b@13000
     /awacs@14000
     /swim3@15000
     /via-cuda@16000
     /adb@0,0
          /keyboard@0,0
          /mouse@1,0
     /54m30@F
/bandit@F40000000
     /53c875@10
/hammerhead@F80000000
```

Listing 6-1 An Open Firmware device tree for the Network Server

Open Firmware starts with the CPU, then examines the memory controller. Then, Open Firmware examines the PCI devices (such as Apple Integrated I/O controller—listed in the example as `gc`) and discovers the available child devices, creating a node for each item it discovers.

Each node in the device tree consists of the following information:

- Property list

  This data structure is part of the node and describes the device. Properties include the name of the device and other device characteristics such as frame buffer size and pixel depth capabilities. Properties are added to the device tree as devices are located.

- Methods

  Device methods are sets of routines used to access the device.

Device tree nodes can also have

- children—other device nodes that are directly below the device in the tree
- a parent—the node that is directly above the device in the device tree.

The following example shows a device tree node and its associated properties. The device is a Symbios Logic 53c875 SCSI adapter card installed in the bottom PCI slot.

```
>dev /bandit@F4000000/53c875@10
>.properties
vendor-id      00001000
device-id      0000000F
revision-id    00000001
class-code     00010000
interrupts     00000001
min-grant      00000008
max-latency    00000040
devsel-speed   00000001
```

```
AAPL,interrupts     0000001D
AAPL,slot-name SLOT6_PCI1
name           53c875
model          NCR,875
compatible     pci1000,f
device_type    scsi
reg  00018000 00000000 00000000 00000000 00000000
     01018010 00000000 00000000 00000000 00000100
     02018014 00000000 00000000 00000000 00000100
     02018018 00000000 00000000 00000000 00001000
     02018030 00000000 00000000 00000000 00008000
power-consumption   007270E0 007270E0
assigned-addresses
     81018010 00000000 00000400 00000000 00000100
     82018014 00000000 F5101000 00000000 00000100
     82018018 00000000 F5100000 00000000 00001000
     82018030 00000000 90000000 00000000 00008000
```

AIX device configuration methods can use the Open Firmware device tree to build their ODM database entries and configure devices. Device methods read the information from a node and pass it to their kernel extension to configure devices. Chapter 7, “Device Configuration for the Network Server,” discusses configuration and the ODM databases; for complete information about the ODM databases, refer to the IBM AIX book *Writing a Device Driver*.

The properties guaranteed to be in the device tree for a device are the `name` property and the `reg` property. The `name` property identifies the device; the name must be stored in the Predefined Devices (PdDv) database of the ODM. The `reg` property contains the address of a device with respect to their parent device. For more information about these properties, see Chapter 7, “Device Configuration With the Network Server.”

Access to the device tree

Your configuration method can access the device tree to discover the properties of a device. By accessing the device tree, your configuration method can find values for a device's properties such as interrupt levels. You need to access the device tree primarily when you configure a device and use a configuration method.

Note: Devices features in the device tree are called properties; features of devices in the ODM are called attributes.

To access information in the device tree, you use Open Firmware routines. In order to reach a specific node, you need the complete path of the node or a "handle" into the device tree. A handle is an index that matches the path of the node in the device tree.

The Open Firmware routines allow you to convert a handle to a path and a path to a handle. See the manual pages in Section IV for more information.

The path of a device

Every configurable hardware device has a path into the Open Firmware device tree. The following example shows a device path:

```
/bandit@F4000000/53c875@10
```

The device path consists of node names separated by slashes (/). The initial slash represents the root node. Each node name has the form

name@address:arguments

You replace the *name* with a text string representing the device and *address* with a hexadecimal number.

When specifying a path, you can include either the *name* or the @*address*, or both. If you include only the name or the address, the firmware selects the device that best matches the information you specified.

The handle attribute

The handle attribute (`OF_handle`) is an index into the Open Firmware device tree that matches the path of the node. The handle can then be used to make further inquiries about properties associated with that node. Each parent configure method sets the handle of its child devices. The handle is a parameter in many of the Open Firmware routines documented in Section 4, *Manual Pages*.

If you know the device path, you can find the handle of a node by using the `path2hdl` routine. For example, to find the handle of the SCSI device with the following path, use this routine:

```
OF_path2hdl ("/bandit@F4000000/53c875@10")
```

For information about defining the handle attribute, see “The Handle Attribute in the Predefined Attributes Database” in Chapter 7, “Device Configuration for the Network Server.”

Open Firmware routines

Configuration methods can obtain device information from the Open Firmware device tree by using the routines in this section.

Note: In most cases, the device-configuration support for the operating system manages most of the interaction between the Open Firmware routines and the device tree. Your device’s configuration methods should communicate with the device configuration subsystem through the Open Firmware routines.

You can use the following routines to traverse the Open Firmware device tree and get information:

- `OF_peer`

  Returns the handle to a device’s sibling (another device with the same parent node) in the device tree.
- `OF_child`

  Returns the handle to a device’s first child.
- `OF_parent`

  Returns the handle to a device’s parent.

- `OF_hdl2path`

  Converts the Open Firmware handle to the path of the device in the Open Firmware device tree.
- `OF_path2hdl`

  Converts the path of the device in the Open Firmware device tree to an Open Firmware handle.
- `OF_getprop`

  Gets the value of an Open Firmware property in the Open Firmware device tree.
- `OF_nextprop`

  Gets the name of the next Open Firmware property.

For complete information about using these routines to get and set information in the Open Firmware device tree, see the manual pages documented in Section 4, *Manual Pages*.

The Open Firmware command-line interface

In most cases, the routines just described are all you need to obtain device information. However, Open Firmware also provides a command-line interface (the Forth monitor) that gives you access to routines for testing devices and examining the device tree. You can only access the Open Firmware environment early in the boot process.

To enter the Open Firmware command-line environment:

1 **Start or restart the system and press Command-Option-OF.**

Continue to hold down the keys until the `0>` prompt appears. The system displays `ok` followed by the `0>` prompt.

If AIX is installed and a root password is set, the security prompt appears, and you need to enter the root password.

2 **If the Security prompt appears, type `login` and press Enter.**

Once you are in the Open Firmware command-line interface, you can display information in the device tree and test devices.

Displaying the device tree

To display the device tree to ensure that your device is listed, enter this command:

```
> dev /
> ls
```

Your Network Server displays a listing similar to the one shown earlier in this chapter in Listing 6-1.

Displaying device properties

To display the properties of a node in the device tree , enter this command:

`> dev` *pathname*

```
> .properties
```

Your Network Server displays a listing similar to the one shown earlier in this chapter in the section “System Startup and Device Discovery.”

7 Device Configuration With the Network Server

The AIX operating system uses the Configuration Manager to define and maintain system configuration information, such as details about peripheral devices. For the Network Server, AIX first obtains this device information from the Open Firmware device tree and stores it in Object Database Manager (ODM) database files. At startup, AIX restores those databases properly and launches the Configuration Manager to load and configure device drivers as necessary. To do this, the Configuration Manager must launch the set of device configuration methods that are associated with each hardware device in the system.

This chapter discusses the configuration methods that you create for each device and highlights the modifications for device configuration to account for hardware and firmware differences between the Network Server and other AIX systems.

The primary differences that configuration methods must take into account are:

- the Network Server uses a different device hierarchy
- the ODM configuration databases have been modified to reflect the new device hierarchy
- the `busresolve` routine is unavailable (Open Firmware provides this information)

The Network Server device hierarchy

Devices are organized into a hierarchical tree of parent-child relationships. Parent device methods detect their children. Once a child device is detected, the AIX Configuration Manager executes the appropriate configuration methods to introduce them to the operating system. These methods load the appropriate device drivers and make the devices available for use. The AIX Configuration Manager oversees the entire configuration process by starting configuration methods, interpreting errors, and managing the configuration of child devices.

The Configuration Manager adds the device information to the ODM device databases by invoking device methods based on device information in the Open Firmware device tree. Configuration is a hierarchical process, starting from the top device node in the ODM databases and descending through all levels of child nodes. Figure 7-1 shows the Network Server device hierarchy.

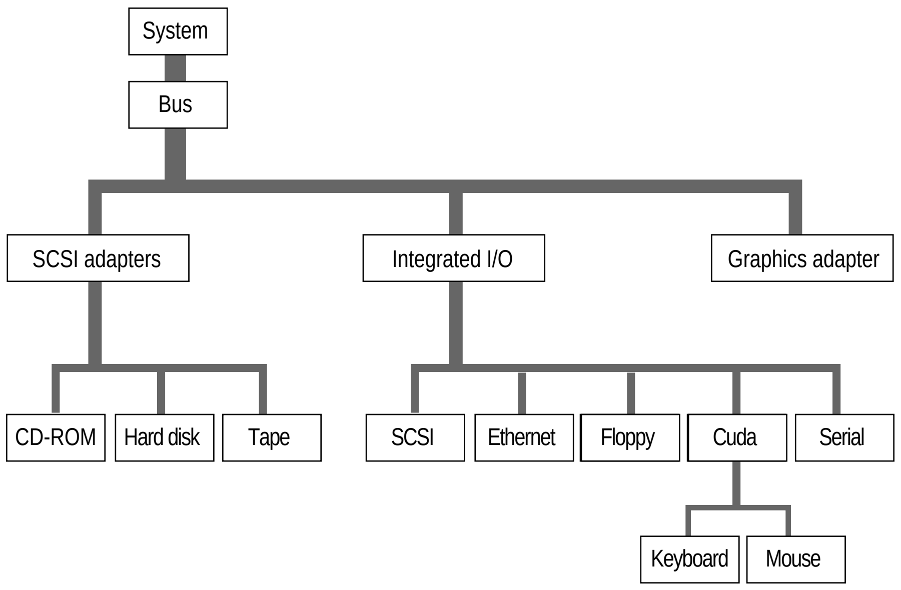


Figure 7-1 The Network Server device hierarchy

Here are some of the changes to configuration routines for the Network Server:

- The `cfgsys` method queries Open Firmware for device tree information.
- The `cfgbus` method for the Network Server supports PCI bus architecture instead of IBM's MicroChannel bus. It queries the Open Firmware device tree to discover its children (such as PCI boards and other buses).
- The `cfggc` method configures the Network Server's Integrated I/O adapter and defines the devices it handles (floppy disk drive, keyboard, mouse, the two serial ports, SCSI, and Ethernet). This replaces IBM's equivalent `cfgsio` subsystem. The `cfggc` methods queries Open Firmware for information about its children.

The ODM databases

The Object Database Manager (ODM) contains system information for AIX. All AIX device configuration information and most of the system configuration information exist as objects (data structures) stored in ODM databases.

Objects that describe devices consist of one or more methods that act upon that object, one or more data fields, and zero or more links. For example, the Predefined Devices (`PdDv`) object class contains seven methods, eighteen data fields, and no links. The methods are `Configure`, `Change`, `Define`, `Start`, `Stop`, `Unconfigure`, and `Undefine`. An object in this class might contain general device information, such as the device driver name and the LED state to display during configuration.

Links between instances of objects reflect dependencies and arrange the objects into a tree structure that determines configuration order and control. Each tree structure constitutes a node. For example, if the SCSI bus adapter needed to be initialized before all the SCSI devices, the SCSI bus object would be placed at the top of the node and thus configured first. All the subsequent SCSI devices linked below it in the node would be configured in turn before the next node was configured.

The ODM provides eight object classes for device configuration. These classes are divided into predefined and customized databases:

- Each predefined database contains a general description of a class of objects. The predefined databases include the Predefined Attributes (`PdAt`) database, the Predefined Connections (`PdCn`) database, and the Predefined Devices (`PdDv`) database.
- Each customized database contains a precise description of an object currently configured with the system (an instance of the predefined description), such as `scsi1` or `hdisk0`. The customized databases include the Customized Devices (`CuDv`) database, the Customized Dependencies (`CuDep`) database, the Customized Attributes (`CuAt`) database, the Customized Device Drivers (`CuDvDr`) database, and the Customized Vital Product Data (`CuVPD`) database.

In AIX for the Network Server, new entries reflecting Apple devices have been added to the ODM databases and other changes have been made for device drivers to support the Network Server platform. These changes are described in the following subsections. (Likewise, many entries in the AIX 4.1 ODM device databases are specific to IBM hardware and these entries do not appear in the ODM databases for the Network Server.)

The device ID field in the Predefined Devices database

AIX for the Network Server uses the device ID field (`devid`) in the Predefined Devices (`PdDv`) database differently than AIX 4.1 for other platforms.

In other versions of AIX 4.1, the value of the devid field often consisted of a pair of POS register values. AIX 4.1 software matched the unique number in the POS registers to the devid field in its PdDv database to determine the type of device. Since this situation is specific to IBM hardware, it does not occur on the Network Server. Instead, AIX for the Network Server matches device information from the Open Firmware device tree to the devid field to identify the device. (The methods that perform this mapping are cfgbus, cfggc, and cfgsys. Each of these methods reads the name and compatible properties from the Open Firmware device tree and searches the ODM for a devid that matches. Open Firmware guarantees a name property for devices in the Open Firmware device tree. If the device does not provide a name during startup, Open Firmware creates a name from the vendor ID and the device ID such as pci1000,3 for the 825A SCSI adapter.)

Here is part of an entry in the PdDv database for the SCSI device used in the examples in Chapter 6, “The Open Firmware Device Tree.”

```
type = "pscsi"
class = "adapter"
subclass = "pci"
prefix = "scsi"
devid = "pci1000,3"
base = 1
uniquetype = "adapter/pci/pscsi"
```

AIX restricts the devid field to 16 characters, including the null terminator. If you wish to store a name in the devid field that is greater than 15 characters plus the NULL terminator, leave the devid field in the PdDv blank to avoid truncation. Instead, use a PdAt attribute=devid and specify type=Z. Store the long name in the deflt field of a PdAt entry. Here is an example definition:

```
PdAt:
     uniquetype =_______
     attribute="devid"
     deflt ="A_long_Open_Firmware_Name"
     value =""
     width =""
     type ="Z"
     generic=""
     rep="s"
     nls_index=""
```

You must fill in the `uniquetype` field with the appropriate *class/subclass/type* identifier associated with your device node—for example, `adapter/pci/pscsi`. The default value should be set to the appropriate string that comes from the Open Firmware `name` and `compatible` properties. It is represented as a string (`rep="s"`) and identified as a type `Z` (`type ="Z"`) attribute.

When the parent configure methods use Open Firmware to discover their children, they will detect the long name and attempt to use the `PdAt` database to map the device.

Building package names for child devices

The `cfgsys`, `cfgbus`, and `cfggc` routines are responsible for building package names for the devices they discover during the installation process. The `cfgbus` routine (responsible for building packages for PCI devices) creates package names with the format *devices.pci.xxxx*, where *xxxx* is the Open Firmware name for the device.

Note: Package names cannot contain a comma (,). When building package names, configuration routines replace commas with plus signs (+) so that the installation works. For example, a package with a device ID of `pci1000,3` maps to a package name of `devices.pci.1000+3`.

The handle attribute in the Predefined Attribute database

The handle attribute (`OF_handle`) is an index into the Open Firmware device tree that matches the path of the node. The handle can then be used to make further inquiries about properties associated with that node. The handle is a parameter in many of the Open Firmware routines documented in Section 4, *Manual Pages*.

Here is a sample definition of the handle attribute in the `PdAt` database:

```
PdAt:
     uniquetype =_______
     attribute="OF_handle"
     deflt ="-1"
     value =""
     width =""
     type ="R"
     generic="D"
     rep="n"
     nls_index=_______
```

You must fill in the `uniquetype` field with the appropriate *class/subclass/type* identifier associated with your device node—for example, `adapter/pci/pscsi`. You should set the default handle to -1. The attribute is also a regular (`type="R"`) attribute represented as a numeric (`rep="n"`). You can set the default handle as a displayable attribute (`generic="D"`) and if so, the `nls_index` should be filled in with the correct message number that maps to the attribute's textual description. For consistency, the textual description is "Open Firmware Device Tree Handle" and must be inserted in your devices catalog source file. (Reminder: The catalog filename and the set number associated with the `nls_index` field are stored in the `PdDv` database.)

The parent device is responsible for storing the Open Firmware handle as an attribute of the child (`OF_handle`) in the ODM databases. This parent sets the handle for the child so the child can access the device tree. The parent device assumes a handle has been predefined in the Predefined Attributes (`PdAt`) database before device configuration and attempts to set its value in the Custom Attributes (`CuAt`) database during configuration. For example, `cfgbus` is responsible for setting the `OF_handle` attribute for its child adapters.

Writing configuration methods

The dynamically loadable and unloadable aspect of the AIX Version 4.1 kernel requires that all device drivers have configuration methods to support the ability to load and unload them from the kernel. Configuration methods are sets of executables including a `Define` method, a `Configure` method, a `Change` method, an `Unconfigure` method, and an `Undefine` method. A `Configure` method is part of a set of configuration methods.

You use `Configure` methods to find a device's attributes, update the values in the ODM, find child devices, and load a device's driver.

When a device's `Configure` method reaches the final task of dealing with its children, it can detect them by traversing the Open Firmware device tree. After the parent `Configure` method defines a child device or even just updates its status in the `CuDv`, the parent must set the child's `OF_handle` attribute in the `CuDv database` if the child device will need to obtain information from the Open Firmware device tree. This must be done each time the system starts up because the `OF_handle` value is not guaranteed to be the same from one startup to the next.

When the `Configure` method for the child device is run, it has access to its own `OF_handle` attribute and can use the Open Firmware routines to search the device tree for device information. (For more information about these routines, see "Open Firmware Routines" in Chapter 6, "The Open Firmware Device Tree," or see the manual pages in the Section IV.)

AIX for the Network Server supports all but one of the device-configuration library routines from other versions of AIX 4.1. The `busresolve` routine is currently not supported; this routine resolves contention of devices on shared system resources such as bus interrupts and DMA channels. On the Network Server, the `busresolve` functionality is replaced by the Open Firmware device tree. Device methods can query the Open Firmware device tree to get device and dbDMA interrupt levels. Open Firmware also provides all assigned device address spaces. AIX for the Network Server provides the following new set of library routines to obtain this information:

```
     resolve_pci_cfg_space    resolve_pci_mem_space
     resolve_pci_io_space     resolve_gc_offset
     resolve_intr_lvl
```

The following sample code uses two of these routines and obtains attributes. The values obtained by this code are used for the sample code at the end of this section.

```
long nodeh;         /*Open Firmware node handle for this device
ulong mem_addr;     /*Open Firmware physical device address*/
ulong mem_addr_size;/*Open Firmware device mapped address space
ulong cfg_addr;     /*Open Firmware configuration space address

/*Read the bus attributes so we can set some of our attributes
 * based on the bus values. */
bat_list = (struct attr_list *) get_attr_list (CuDv_bus.name,
                    CuDv_bus.PdDvLn_Lvalue, &scratch, 2);
if (bat_list == (struct attr_list *) NULL)
{
     DEBUG_1 ("bld_dds: get_attr_list for parent failed with
          error %d\n",scratch)
     return (scratch);
     /*scratch contains the error from get_attr_list*/
```

```
}
/* Use the getatt function to extract the proper
 * customized or predefined attribute values from the ODM
 * database. Use the bus attribute list to get the bus
 * base_addr */
if ((rc = getatt(bat_list, "base_addr", (void *) &dds.base_addr
               'i', &scratch)) !=0)
{
     return (rc);
}
DEBUG_1 ("dds.base_addr=%x\n",dds.base_addr)
/* Use the bus attribute list to get the bus-range for the bus
if ((rc = getatt (bat_list, "bus-range", (void *)
          &dds.parent_pci_bus,
               'i', &scratch)) !=0)
{
     return (rc);
}
DEBUG_1 ("dds.base-range=%x\n",dds.base-range)

rc = get_OF_status(nodeh);
if (rc !=E_OK) {
     return (rc);
}
rc = resolve_pci_mem_space (nodeh, &mem_addr, &mem_addr_size);
if (rc == E_OK) {
     dds.mem_space_addr = mem_addr;
     dds.mem_space_length = mem_addr_size;
} else {
     return (rc);
}
rc = resolve_pci_cfg_space (nodeh, &cfg_addr);
if (rc == E_OK) {
     dds.cfg_space_reg_val = cfg_addr;
     return (rc);
}
```

The routines that obtain device information use a basic set of library routines for querying the Open Firmware device tree including the following:

```
     OF_child       OF_peer        OF_parent
     OF_hdl2path    OF_path2hdl    OF_getprop
     OF_nextprop
```

The complete reference pages for these routines are in Section IV, *Man Pages*.

The `subclass` field in the PdDv database should contain the string `pci` for any new PCI device.

PCI device developers need to obtain certain attributes to use with kernel services. When a device driver registers an interrupt handler, an interrupt structure must be filled. Use these guidelines for `struct intr` from `intr.h`.

Interrupt structure field	**Source of data**
bid	Parent bus `bus_id` attribute
bus_type	Parent bus `bus_type` attribute
level	Use `resolve_intr_lvl`() routine (See man page in section 4.)
priority	Device's PdAt `intr_priority` attribute

Your configuration method determines these values.

When a device driver needs to use the `pci_cfgrw()` kernel routine, the `mdio` structure must be filled. Use these guidelines for MACH_DD_IO from `mdio.h`. The `pci_cfgrw` routine is documented in Section IV, *Manual Pages*.

mdio Field	Source of data
`md_sla`	Use `resolve_pci_cfg_space()` routine to pass in appropriate value. The value represents: `(bus# * 16) + (devnum * 8) + function)`
`md_length`	Specifies the `bus-range` attribute of the parent bus or parent bridge (whichever applies). Retrieve this attribute through the configure method `bus-range` attribute.
`md_data`	Pointer to data buffer.
`md_size`	Specifies the number of items of size specified by the `md_incr` parameter. The maximum size is 256.
`md_incr`	Specifies the access type. MV_WORD is the only supported type for the Network Server.
`md_addr`	This should be the base address of bus device ORed with the configuration register address. The base address portion should come from the Parent bus `base_addr` attribute.

Typically, you obtain this data with the configuration method, initialize the data with the device's DDS structure, and pass the information to the driver.

Here is some sample code that manages PCI configuration register cycles. (This code uses the sample code earlier in this chapter to obtain values.)

```
#include <sys/mdio.h>     /*for config  cycles*/
struct LSA_def *lsa;     /*overall adapter instance struct */
struct mdio md;          /*from mdio.h*/
ulong cfg_data;
```

```
/*
*  This value is retrieved by the config method through
*  the base_addr attribute;
*  lsa->ddi.base_addr;        pci bridge device base address

*  This value is retrieved by the config method through the
*  resolve_pci_cfg_space routine. (See the man page for details
*  lsa->ddi.cfg_space_reg_val; bus/device/function combination

*  This value is retrieved by the config method through the
*  bus range attribute:
*  lsa->ddi.parent_pci_bus;   parent device PCI bus number

*  Initialize the mdio struct used for pci_cfgrw()
*/
md.md_addr = lsa->ddi.base_addr;
/*pci bridge device base address*/
/* OR in the particular config register wanted,
 * such as its address */
md.md_addr |= 0x0;
               /* go for config reg 0 for this case */
md.md_sla = lsa->ddi.cfg_space_reg_val;
md.md_data = &cfg_data;  /* where pci_cfgrw puts the data */
md.md_size = 1;          /* read 1 datum of size md_incr */
md.md_incr = MV_WORD;
                    /*config data size will be int (4 bytes) */
md.md_length = lsa->ddi.parent_pci_bus;
                    /*parent’s PCI bus number */
```

```
/* read config register 0 */
rc = pci_cfgrw(0, &md, READ);
                    /*pci_cfgrw() is a kernel routine */
if ((cfg_data & 0xF0000) == 0xF0000 {
     printf("SymLogic875 config reg 0:dev/id = %x\n",cfg_data);
     lsa->chip_type = 0x875;
}
if (rc == 0) { /* if no previous error */
     md.md_addr = lsa->ddi.base_addr;
               /*bridge device base address*/
               /*OR in the particular config register wanted */
     md.md_addr |= 0x4;
               /* go for config reg 4, Status/Command reg */
     rc = pci_cfgrw(0, &md, READ);
     printf("Config Reg 4: Status/Cmd; initially = %x\n",cfg_da
}
if (rc == 0) { /*if no previous error */
     /*now let's write our desired values */
     cfg_data = 0x16; /*BUS MASTER, MEM I/O Space, MEM WR & INV
     rc = pci_cfgrw(0, &md, WRITE);
}
if (rc == 0) { /*if no previous error */
     /* read it back to verify */
     rc = pci_cfgrw(0, &md, READ);
     printf("Config Reg 4: changed to %x\n",cfg_data);
}
```

8 Device I/O on the Network Server

Even though a driver can perform many tasks, you usually write a driver to output data to a device or demand data from a device; in other words, drivers usually perform device I/O. A driver may have to read from, or write to, registers on a card that serves as an adapter between an I/O bus and a device connected to the card, or the driver may have to set up the means for data to be transferred in some other way.

The primary I/O differences between AIX for the Network Server and AIX 4.1 for other platforms are as follows:

- the Network Server does not use the IOCC and MicroChannel Adapter (MCA) architectures to control I/O.
- the Network Server uses different memory mapping for I/O spaces.
- some components of the Network Server use descriptor-based direct memory access (dbDMA) and the Network Server does not implement central DMA support and the DMA support services.

This chapter highlights the differences between AIX for the Network Server and AIX 4.1 for IBM platforms and discusses the following device I/O topics:

- how the Network Server translates effective addresses to real addresses
- how the Network Server I/O controller maps I/O space
- how programmed I/O works with PCI devices
- how to use DMA with the Network Server
- how to allocate contiguous physical memory for a device driver

Most of the differences related to the PCI architecture are hidden in the driver configuration methods and the `iomem_att` and `iomem_det` primitives. These AIX functions allow the driver to map and unmap I/O address space from the operating system.

Address translation on the Network Server

Device drivers need to map addresses from physical memory to the virtual memory of the kernel and to the virtual memory of certain processes. To do so, drivers must perform address translation.

When a device driver gains control of the system, it uses routines in the kernel to get to PCI memory space in the physical memory. The `iomem_att` and `iomem_det` routines perform mapping address translation for the driver.

AIX for the Network Server implements block address translation and segment address translation for device I/O access. Figure 8-1 shows how block address translation maps effective addresses to real addresses.

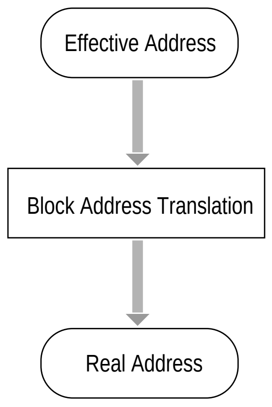


Figure 8-1 Block address translation

Block address translation (BAT) is an alternative to page address translation. AIX for the Network Server uses BAT registers to map I/O. A BAT register can map up to 256 MB of information. The `iomem_attach` and `iomem_detach` routines hide the details of this address space from the device driver.

For more information on block address translation, see the book *PowerPC Architecture*.

I/O controller types on the Network Server

The Network Server processor architecture expects an I/O controller to provide an interface between the system bus (the one the processors use to access RAM) and an I/O bus. A computer system may have more than one I/O bus, but each bus has its own I/O controller.

The Network Server has the following processor type, I/O controllers, and I/O bus protocol.

Processor	Controller	Bus protocol	Address space for I/O
PowerPC	PCI bridge	PCI	Real address (memory-mapped I/O)

The I/O space is part of real address space—in other words, a memory controller translates real addresses so that certain address ranges access system RAM, and other address ranges access other devices. In this sense, I/O space is memory-mapped. For implementation details, see *Setting Up the Network Server*.

When you use the `iomem_att` routine to establish access to memory mapped I/O, you need to provide the routine the values for the I/O controller and the I/O bus protocol.

I/O space on the Network Server

Device drivers use the `iomem_att` routine to access physical memory. The memory controller maps I/O space. You can access I/O space by generating real addresses through address translation.

Here are the main address allocations.

Address	Purpose
0x00000000–0x7FFFFFFF	RAM (2 GB)
0x80000000–0xF8FFFFFF	PCI memory
0xF0000000–0xF8FFFFFFF	PCI bridges/system control
0xF9000000–0xFEFFFFFFFF	PCI memory
0xFF000000–0xFFFFFFFFF	ROM

Figure 8-2 shows the address map. The first bar represents the first part of an address; the second bar represents the second part.

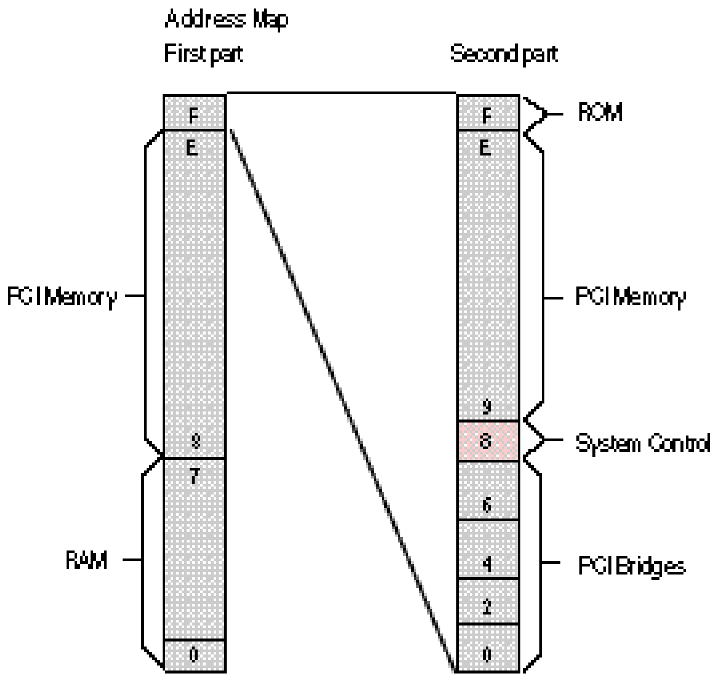


Figure 8-2 Address mapping

Programmed I/O to PCI devices

A routine performs programmed I/O whenever it issues a load or store instruction with an address mapped to a bus or device. You distinguish programmed I/O, where a system processor performs the data transfer, from direct memory access (DMA), where data is transferred by some other means.

The following code sample uses programmed I/O and the `iomem_att` and `iomem_det` routines:

```
#include <sys/ioacc.h>
#include <sys/adspace.h>
int
read_reg(struct LSA_def * lsa,
/* lsa is overall driver struct */
     uint offset)
{
     uint ret_code = 0;
     uint val;
     uint* addr;

     void*  mac_io_addr;   /*keep intact for io_det */
     struct  io_map iom = {IO_MEM_MAP, IOM_INHIBIT,
          SEGSIZE/2, REALMEM_BID, O};
     uchar*    chip_addr;

     iom.busaddr = (struct ipl_cb *) lsa->chip_base_raddr;
          /*chip real addr */
     mac_io_addr = iomem_att (&iom);
     chip_addr = (uchar *) mac_io_addr;
     addr = (uint*) (chip_addr + offset);
     val = *addr;
     ret_code = word_revers(val);
     /*byte swap needed for this driver */
     iomem_det (mac_io_addr);
     return (ret_code);
}
```

The argument to `iomem_att` is a pointer to an `io_map` structure as defined in `sys/ioacc.h`. The calling routine provides the size of the address space needed, a bus ID, which specifies the bus type of the region to be mapped. Network Server drivers use REALMEM_BID as the bus ID.

A call to `iomem_att` returns a valid virtual address. This address is to be passed to `iomem_det` after the I/O operation is complete.

Direct memory access

To transfer data to or from a device without having a system processor issue load or store instructions, you can use direct memory access (DMA), which relies on capabilities designed into the Network Server DMA controller and the adapter communicating with the attached device.

AIX 4.1 for IBM platforms has a DMA subsystem that uses a central DMA engine and relies on IOCC and MCA. AIX for the Network Server, however, does not support central DMA services. Each device determines how to use DMA services.

There are two types of DMA:

- DMA master

  When an adapter card arbitrates for the bus and is able to transfer data directly by generating its own bus addresses and transfer lengths, then the transfer is a DMA master operation, and the card is a DMA master adapter.
- DMA slave

When an adapter card arbitrates for control for the bus but lacks the ability to generate its own bus addresses to transfer data, a third party (a DMA controller) must perform the data transfer. In this case, the transfer is a DMA slave operation, and the card is a DMA slave adapter.

Because bus addresses are meaningless during DMA slave operations, DMA slave adapters cannot use the addresses during the DMA transfer to indicate the intended location for the data. However, the programmed I/O commands that the device driver has previously issued to the adapter typically enable the adapter to know where the data is to be put or where it is to be retrieved.

Allocating contiguous physical memory

Some device drivers need to allocate contiguous physical memory space. Because of this, AIX for the Network Server uses IBM's *rheap* kernel code available on their RSPC class boxes. AIX for the Network Server lets the user configure the size of the real heap. (With IBM, the heap is a fixed size of 128 KB.)

The Network Server implementation of the real heap includes the following features:

- A `real_heap_size` attribute was added to `sys0` in the PdAt database.
- AIX for the Network Servers includes database entries in SMIT to let you set the size attribute.
- The `mkboot` command (invoked by `bosboot`) has a new function, `get_heapsize()`, which uses `getattr()` to obtain the value of `real_heap_size`.
- The `mkboot` command saves the heap size into the `o_resv2` field of the `xcoff` header of the generated boot image.
- The `bootapple` command uses the value in `o_resv2` and allocates contiguous physical memory right below the newly generated IPLCB. The `init_rheap_buc()` routine finds or generates an entry in the Bus Unit Controller table in the IPLCB which describes the allocated real heap memory.

9 The Network Server Interrupt Subsystem

Because interrupt hardware for the Network Server is different from the interrupt hardware of other platforms supporting AIX 4.1, the interrupt subsystem has been modified. This chapter provides an overview of the Network Server interrupt subsystem, highlighting the differences in interrupt levels and mapping.

Overview of the interrupt subsystem

Adapters on any bus can generate interrupts to the host processor. Each interrupt is associated with a particular level, which the processor uses to determine how to handle the interrupt. Interrupt levels for the Network Server can be shared—that is, more than one device can generate interrupts on the same level.

Device drivers provide an interrupt handler to which the system dispatcher transfers control of an interrupt. An interrupt handler is a routine that is called by the kernel whenever an interrupt occurs at a given level. The interrupt handler must first determine whether the interrupt was caused by the adapter the driver is managing. If it was not, the handler exits immediately and returns and indication of failure. If it was, the interrupt handler performs the processing that is needed to deal with the interrupt, resets the interrupt in the adapter, and returns to the kernel.

When a device configures itself, it specifies the priority of interrupts from its associated adapter. When an interrupt occurs, interrupts from other devices at that priority level and below are disabled. Higher priority interrupts can still occur. Interrupt handlers for low-priority devices (such as printers) can be preempted if an interrupt occurs on a high-priority device.

When Open Firmware queries a device for information during the start up process, it obtains the interrupt level and includes it in the device tree. To obtain this information from the device tree, you use the Open Firmware routines. For information about these routines, see Chapter 6, “The Open Firmware Device Tree” and the manual pages in section 4.

Interrupt levels

To determine the source of an interrupt, the Network Server processor depends on a software interrupt level—a value that corresponds to a distinct hardware interrupt source on a device. When the processor receives an interrupt, the processor determines the interrupt level and uses it to call the appropriate interrupt handler.

You can obtain the interrupt value for a device from the Open Firmware device tree by using the `resolve_intr_lvl()` routine. See the man page in Section IV for complete information about this routine. The value of the Apple interrupt property must be used as the `level` field in the `intr` structure passed to `i_init`().

The following table summarizes the interrupt levels for different types of buses.

Bus	Interrupt sensitivity	Can interrupt levels be shared ?	Are interrupt levels programmable?	Available interrupt levels
MCA	Level	Yes, with same priority	Yes	1–16
PCI	Level	Yes	Yes	15
ISA	Edge	No	No	5, 7, 9, 11, 14, 15
Apple PCI	Level	Yes	No	Determined by Open Firmware

The AIX 4.1 operating system uses the `busresolve` routine to find a bus interrupt level. AIX for the Network Server, however, does not support the `busresolve` routine; its functionality is provided by Open Firmware. During the startup process, Open Firmware obtains a device's interrupt level and stores the value in the Open Firmware device tree. You can use the routines described in Chapter 6, "The Open Firmware Device Tree," to access this value.

Note: The mapping of an interrupt's source to a specific processor interrupt level depends on the hardware architecture. Open Firmware handles the mapping of interrupts so that you do not have to.

10 Implementing Graphics Input and 2D Graphics Device Drivers

The graphics system for the Network Server has only a few changes from the AIX 4.1 system for other platforms. The primary differences are as follows:

- The device-dependent level of the X Server is different to support the Network Server. This change does not affect device drivers.
- The Network Server does not support 3D features.

  The 3D features allow you to provide lighting and shading. Unsupported 3D features include graPHIGS, PEX, 3D libraries, and GAI 3D Model libraries.

This chapter describes the X Server and changes to support for input and display devices.

Graphics environment

To provide a graphics environment, AIX for the Network Server provides only the 2D portions of AIX Windows. The 2D portions enable the user to run system management applications or any other X Window System™ or Motif applications. AIX for the Network Server does not provide the 3D portions of AIX, such as PEX (the PHIGS extension to X Window System) or Silicon Graphics' OpenGL.

AIX Windows consists of the X Window System (X11R5 server, client libraries and applications, fonts, include files, and so on), Motif 1.2.3 (libraries, include files, and applications), and the COSE Desktop.

The X server

The X server provides a device-dependent layer that isolates nearly all hardware dependencies. AIX for the Network Server has changed much of this layer to work with Apple's video drivers.

- The Network Server supports the video modes provided by the video chip on the main logic board for 8-bit, 256 color displays. The Network Server meets the SVGA standards.
- The underlying video driver in the kernel has been changed to support the video chip. It interacts with the chip just like other video adapters in AIX.
- To support color with the X server, AIX for the Network Server includes the standard X Window System routines to calculate the clipping area. (AIX 4.1 does not include these routines because the IBM hardware provides this functionality.)

Mouse support

X applications typically expect a three-button mouse. Because the Network Server supports the Apple one-button mouse, AIX for the Network Server maps the arrow keys to the middle and right mouse buttons to overcome the limitations of the one-button mouse. With three-button mouse devices, the arrow keys perform normally.

Other input device issues

In addition, the X server’s key mappings have also been modified to support Apple’s keyboards. The X Server has a keymap structure to reflect the keys on a keyboard. AIX for the Network Server has a keymap to reflect Apple keyboards.

The `xinitrc` startup script has been modified to reflect Apple ISO keyboards for foreign languages.

Section III Using the AppleTalk API

This section describes the programming interfaces to the AppleTalk API.

11 AppleTalk Programming Interfaces

This section describes the programming interfaces of the different AppleTalk Stack modules. The access to the following modules (DDP, ATP, NBP, RTMP, PAP, ASP) is provided through library function calls.

Datagram Delivery Protocol (DDP)

The Datagram Delivery Protocol (DDP)is a best-effort protocol, meaning there is no guarantee of reliable datagram delivery.

DDP is a connectionless datagram protocol, providing for delivery of datagrams not only from node to node but also from AppleTalk socket to AppleTalk socket and from internet to internet. You establish an application's connection to DDP by calling `ddp_open()` routine using one of the 254 AppleTalk sockets that the DDP layer provides on each node.

The `ddp_open()` routine takes an AppleTalk socket number as its argument. If a number between 1 and 127 (inclusive) is supplied, DDP returns a connection to that AppleTalk socket, known as a *static AppleTalk socket*. If the number 0 is supplied, DDP return a connection to an AppleTalk socket number in the range of 128 through 254, known as a *dynamic AppleTalk socket.*

Dynamic AppleTalk sockets are the most commonly used socket type. When you open a dynamic socket, you are guaranteed a unique socket ID, but you do not specify the number of that socket.

For certain applications, only static sockets are appropriate. For example, certain AppleTalk sockets are reserved for clients such as the AppleTalk core protocols (for example, ATP, NBP, and RTMP) and for low-level network services such as echoers. Others are reserved for unrestricted experimental use. These experimental sockets are not recommended for commercial products because there could be conflicting usage of the same socket numbers by different developers.

A datagram consists of a DDP header followed by data. The header contains:

- the data length
- source and destination
- addresses (The AppleTalk addresses in the header contain the full internet addresses of the AppleTalk sockets.)
- protocol type that identifies the caller

- a hop count field that keeps track of the number of routers visited to avoid transmitting the datagram indefinitely
- a checksum field

On an AppleTalk internet, the router determines the route the datagram takes from source to destination. See *Inside AppleTalk* for more information about datagrams.

DDP library functions

The DDP functions use this structure:

```
typedef struct {
     u_short unused : 2,
               hopcount : 4,
            length : 10;
     u_short checksum;
     at_net dst_net;
     at_net src_net;
     at_node dst_node;
     at_node src_node;
     at_socket dst_socket;
     at_socket src_socket;
     u_char type;
     u_char data[DDP_DATA_SIZE];
} at_ddp_t;
```

The `checksum`, `dst_net`, `dst_node`, `dst_socket`, and `data` members must be set before you can write the datagram. The remaining members are set by DDP. The `length` member specifies the DDP packet length. If the `checksum` member is nonzero, DDP calculates a checksum for the data member and stores it in `checksum`.

The following sections detail the DDP functions.

The `ddp_open` function

```
int ddp_open (socket)
```

```
at_socket *socket;
```

This function opens a static or dynamic AppleTalk socket for sending or receiving DDP datagrams and returns a file descriptor that can be used to read and write DDP datagrams. The *socket* parameter is a pointer to a DDP socket number to open. If the socket number is 0 or if *socket* is NULL, a DDP socket is assigned dynamically. Otherwise, the static socket (in the range 1 to 127, inclusive) specified by *socket* is opened. Once opened, you can send and receive datagrams by calling the UNIX `read`() and `write`() system calls. You can send and receive asynchronously by calling the UNIX `select`() system call and by calling `fcntl`() with the `O_NDELAY` parameter. If you want to provide an implementation of ATP or ADSP directly over Apple's DDP layer, you must use `atpproto_open` () or `adspproto_open` , respectively.

The `ddp_close` function

```
int ddp_close(fd)
```

```
int fd:
```

This function closes the DDP AppleTalk socket identified by *fd*.

The `atpproto_open` function

```
int atpproto_open (socket)
```

```
at_socket *socket;
```

This function must be used, in place of the `ddp_open`() function, by an ATP layer running over DDP. It opens a static or dynamic DDP socket for sending or receiving ATP packets and returns a file descriptor that can be used to read and write DDP datagrams containing ATP protocol data units. The *socket* parameter is a pointer to a DDP socket number to open. If the socket number is 0 or if *socket* is NULL, a DDP socket is assigned dynamically. Otherwise, the static socket (in the range 1 to 127, inclusive) specified by *socket* is opened. Once opened, you can send and receive datagrams by calling the UNIX `read`() and `write`() system calls. You can send and receive asynchronously by calling the UNIX `select`() system call and by calling `fcntl`() with the `O_NDELAY` parameter.

The `adspproto_open` function

`int adspproto_open (`*`socket`*`)`

`at_socket *`*`socket`*`;`

This function must be used, in place of the `ddp_open`() function, by an ADSP layer running over DDP. It opens a static or dynamic DDP socket for sending or receiving ADSP packets and returns a file descriptor that can be used to read and write DDP datagrams containing ADSP protocol data units. The *socket* parameter is a pointer to a DDP socket number to open. If the socket number is 0 or if *socket* is NULL, a DDP socket is assigned dynamically. Otherwise, the static socket (in the range 1 to 127, inclusive) specified by *socket* is opened. Once opened, you can send and receive datagrams by calling the UNIX `read`() and `write`() system calls. You can send and receive asynchronously by calling the UNIX `select`() system call and by calling fcntl() with the `O_NDELAY` parameter.

DDP Error Codes

All of the DDP functions return –1 to indicate an error with one of these error codes in `errno`:

EACCES

A user who does not have permission attempted to open a static AppleTalk socket.

EADDRINUSE

An attempt was made to open a specific static socket that is already in use or an attempt was made to open a dynamic socket, all of which are in use.

EINVAL

An attempt was made to open an invalid AppleTalk socket number.

EMSGSIZE

A datagram is too large or too small.

ENETDOWN

The network interface is down.

ENOBUFS

DDP is out of buffers.

Routing Table Maintenance Protocol (RTMP)

RTMP creates (only in Routers) and maintains the routing information necessary to route packets to their destinations. In AIX on nonrouter nodes, RTMP is used primarily for maintaining network addresses when routers are present. When a router is present, it periodically transmits the current network number and router number. This allows for proper routing of a user's request by means of the AppleTalk internet address.

RTMP library functions

The only RTMP library function is `rtmp_netinfo`(). Applications that use `rtmp_netinfo`() should include the `appletalk.h` header file.

```
int rtmp_netinfo(fd, addr, bridge)
int fd;
at_inet_t *addr, *bridge;
```

This function obtains node and bridge addresses. The parameter *fd* specifies an AppleTalk file descriptor for an AppleTalk socket. The *addr* and *bridge* parameters point to `at_inet_t` structures, which are used to store the 32-bit address of a NVE (Network Visible Entity).

To get the address of a node that is associated with an AppleTalk file descriptor, set *fd* to the value of a valid AppleTalk file descriptor and set the value of *bridge* to NULL. On return, the members of the `at_inet_t` structure pointed to by *addr* contain these values:

`net`	The network number.
`node`	The node number.
`socket`	The socket number.

A *network number* is between 1 and 65535. The network number is supplied by a router node and is not built into the nodes on a network. Therefore, if there are no router nodes on the network, the returned number is 0 for LocalTalk networks. For Ethernet networks, the network number is the network number the node had before the router disappeared, or a value from the startup range. Node numbers are assigned dynamically by the LAP layer when the network is established. A *node number* is between 1 and 254.

To get the address of a *bridge*—a device that provides the signal amplification necessary to extend a cable—set *fd* to –1. On return, the members of the `at_inet_t` structure pointed to by *addr* contain values for the local address and the members of the `at_inet_tr` structure pointed to by *bridge* contain these values:

`net`	For LocalTalk networks, 0; for EtherTalk networks, the network number of the local router.
`node`	The node number.
`socket`	Because the socket member for a bridge has no meaning, this member is 0.

The `rtmp_netinfo`() function returns –1 to indicate an error with this error code in `errno`:

EINVAL	Indicates that both *addr* and *bridge* are NULL.

For additional values of `errno`, see “Datagram Delivery Protocol (DDP),” earlier in this section.

AppleTalk Transaction Protocol

The AppleTalk Transaction Protocol (ATP) provides reliable transport between two AppleTalk nodes. At the ATP level, reliability is provided by these methods:

- noting lost packets and resending them
- noticing duplicate packets and discarding them
- detecting out-of-sequence packets and reassembling them in order

Reliability is achieved through the use of exactly-once (XO) transactions. With at-least-once (ALO) transactions, some of this reliability can be lost because ATP does not guarantee filtering of duplicate requests in all situations (for example, if a router fails during a transaction). See *Inside AppleTalk* for details.

In the current release, the ATP library functions provide both synchronous and asynchronous operations with the interface. ATP library functions allow for the preservation of packet boundaries. If your packets are of variable size or mixed composition (headers in one, data in another), you must preserve packet boundaries.

There are six functional areas of software support for ATP under AIX. One function is provided for each of these services:

- AppleTalk socket acquisition (`atp_open`())
- AppleTalk socket disposal (`atp_close`())
- request receipt (`atp_getreq`())
- response sending (`atp_sendrsp`())
- request sending (`atp_sendreq`())
- response receipt (`atp_getresp`())

ATP library functions

The ATP functions use the following structures.

```
typedef struct at_retry {
     short interval;
     short retries;
     u_char backoff;
} at_retry_t;
```

This structure specifies the retry interval and maximum number of retries for an ATP transaction. The `interval` member specifies the number of seconds to wait before retrying an ATP request. The `retries` member specifies the maximum number of retries for an ATP request. The `backup` member specifies the value by which the retry interval is to be multiplied, to a maximum of 16 seconds, on each retry.

```
typedef struct at_resp {
     uchar bitmap;
     struct iovec resp[ATP_TRESP_MAX];
     int userdata[ATP_TRESP_MAX];
} at_resp;
```

This structure specifies buffers to be used for response data. The `bitmap` member specifies the buffers for which responses are expected. The `resp` member is an `iovec` structure that describes the response buffers and their lengths. The `userdata` buffer member is an array of 32-bit words that holds the user bytes for each ATP response.

All of the ATP functions return –1 to indicate an error and one of the following error codes in `errno`:

EAGAIN

Indicates that the request failed due to a temporary resource limitation.

EINVAL

Indicates that a *dest*, *len*, *resp*, or *retry* parameter is invalid.

ENOENT

Indicates that the request was an attempt to send a response to a nonexistent transaction.

ETIMEDOUT

Indicates that the request exceeded the maximum number of retries.

EMSGSIZE

For `atp_sendreq`(), indicates that the response was larger than the buffer or that more responses were received than expected. The response is truncated to the available buffer space.

For `atp_getreq`(), indicates that the request buffer is too small for the requested data. The requested data is truncated is the available buffer space.

For `atp_sendrsp`(), indicates that the response is too large. The maximum in bytes is ATP_DATA_SIZE.

The following sections detail the ATP functions.

The `atp_open` function

```
int atp_open(socket)
```

```
at_socket *socket;
```

This function opens an ATP AppleTalk socket. It returns a file descriptor that you use when you call other ATP functions. The *socket* parameter is a pointer to an ATP socket number to open. If the socket number is 0 or if *socket* is NULL, an ATP socket is assigned dynamically. Otherwise, the static socket (in the range 1 to 127, inclusive) specified by *socket* is opened.

The `atp_close` function

```
int atp_close(fd)
```

```
int fd;
```

This function closes the AppleTalk socket specified by *fd*.

The `atp_sendreq` function

```
int atp_sendreq(fd, dest, buf, len, userdata, xo, xo_relt,
               tid, resp, retry, nowait)
```

```
int fd;
```

```
at_inet_t *dest;
```

```
char *buf;
```

```
int len, userdata, xo, xo_relt;
u_short *tid;
at_resp_t *resp;
at_retry_t *retry;
int nowait;
```

This function sends an ATP request to another socket. If the flag `nowait` is set to non-zero, this functions returns immediately without waiting for the response (the response must later be received using the `atp_getresp`() function); otherwise, it waits for the response. The parameter *fd* specifies the file descriptor to use in sending the request, and *dest* specifies the AppleTalk internet address of the AppleTalk socket to which the request is to be sent. The parameter *buf* points to the data to be sent and *len* specifies the number of bytes to send. The parameter *userdata* specifies the user bytes for the ATP request header.

The parameter *xo* is TRUE if the request is to be sent exactly once or FALSE if the request can be sent more than once. Setting *xo* to FALSE causes the next parameter, *xo_relt* (which should be 0) to be ignored. When *xo* is TRUE, *xo_relt* is used to set the release timer value on the remote socket. The value of *xo_relt* can be 0 (or `ATP_XO_DEF_REL_TIME`) for the default value, `ATP_XO_30SEC`, `ATP_XO_1MIN`, `ATP_XO_2MIN`, `ATP_XO_4MIN`, or `ATP_XO_8MIN`.

On return, the parameter *tid* stores the transaction identifier for this transaction. The parameter *resp* is a pointer to an `at_resp_at` structure that, on return, stores the response for this transaction. The length of each response buffer, specified by the `iov_len` member, is updated on return from `atp_sendreq`(). When you use the `atp_sendreq`() function in the no-wait mode, you do not need to specify the *resp* structure meaning the *resp* parameter can be 0; the default bitmap will be 0xff. If a different bitmap is desired, *resp* can be specified with only a bitmap value; the rest of the parameters in the *resp* structure are not necessary and need only be specified during the `atp_getresp`() function.

The parameter *retry* is a pointer to an `at_retry_t` structure, which specifies the retry interval and the maximum number of retries for this ATP transaction. If the value of *retry* is NULL, the retry interval is `ATP_DEF_INTERVAL` (that is, 2 seconds), the maximum number of retries is `ATP_DEF_RETRIES` (that is, 8) and the backoff value is 1. Setting the retries member of *retry* to `AT_INF_RETRY` causes the transaction to be repeated indefinitely.

The parameter *nowait* should be set to 0 to indicate synchronous operation or to 1 to indicate no-wait operation.

The `atp_getreq` function

```
int atp_getreq(fd, src, buf, len, userdata, xo, tid,
               bitmap, nowait)

int fd;

at_inet_t *src;

char *buf;

int *len, *userdata, *xo;

u_short *tid;

u_char bitmap;

int nowait;
```

This function receives an incoming ATP request sent from another AppleTalk socket. The parameter *fd* specifies the file descriptor to use in receiving the request, and *src* specifies the AppleTalk internet address of the AppleTalk socket from which the request was sent. The parameter *buf* points to a buffer where the incoming data will be stored, and *len* specifies in bytes the length of *buf*.

On return, the parameter *userdata* contains the user bytes from the ATP request header; the parameter *xo* is TRUE if the request is to be sent exactly once or FALSE if the request can be sent more than once; the parameter *tid* stores the transaction identifier for this transaction; and the parameter *bitmap* contains the responses expected by the requester.

In no-wait operation, this function returns -1 immediately if there is currently no ATP request from another AppleTalk socket.

The parameter *no-wait* should be set to 0 to indicate synchronous operation or to 1 to indicate no-wait operation.

The `atp_sendrsp` function

```
int atp_sendrsp(fd, dest, xo, tid, resp)

int fd;

at_inet_t *dest;

int xo;

u_short tid;

at_resp_t *resp;
```

This function sends an ATP response packet to another AppleTalk server socket. The parameter *fd* specifies the file descriptor to use in sending the response, and *dest* specifies the AppleTalk internet address of the AppleTalk socket to which the response should be sent. The value of *tid* is the transaction identifier for this transaction. The parameter *resp* is a pointer to an `at_resp_t` structure that contains two arrays for the response data: `resp`, which is an eight-entry `iovec` array and `userdata`, which is an eight-entry array. The `iov_base` member of each `iovec` member points to a buffer containing the response data. The `iov_len` member specifies the length of each response buffer. Each `userdata` member contains the user data to be sent with its respective ATP response packet.

The `atp_getresp` function

```
int atp_getresp(fd, tid, resp);

int fd;

u_short tid;

at_resp_t *resp;
```

This function is used in the no-wait mode, (the `atp_sendreq`() function was called without waiting for the response) to receive the incoming ATP response sent from another AppleTalk socket. The parameter *fd* specifies the file descriptor to use in receiving the response.

On return, the parameter *tid* stores the transaction identifier for the previous transaction request for the matched response. The parameter *resp* is a pointer to an `at_resp_t` structure that, on return, stores the response information for the transaction.

The `atp_look` function

```
int atp_look(fd);
int fd;
```

This function looks at the current event on an ATP endpoint. It returns a 0 if the event is a transaction request coming from another ATP socket; it returns a non-zero value if the event is a transaction response or completion of a previous synchronous transaction request.

The `atp_abort` function

```
int atp_abort(fd, dest, tid);
int fd;
at_inet_t *dest;
u_short tid;
```

This function aborts a transaction request or transaction response in progress. If the parameter `dest` is set to 0, it is a transaction request abort. The value of *tid* is the transaction identifier of the transaction in progress.

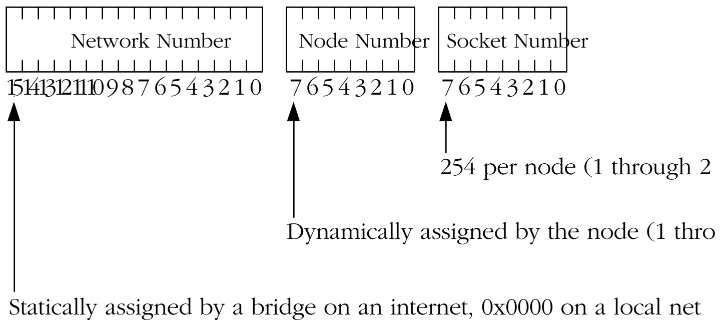
On success, this function returns 0; otherwise, it returns -1.

Name Binding Protocol

The Name Binding Protocol (NBP) translates the character name associated with a *network-visible entity* (NVE) into an AppleTalk internet address for use with the AppleTalk protocols. NBP allows the use of names rather than the numeric identifiers required for communication over the AppleTalk network. The address of a named entity must be obtained before that entity can be used over AppleTalk. The process of obtaining the address is known as *name binding*. NBP allows special characters to be substituted (wildcard name look-ups) in place of strings.

The AppleTalk protocols refer to an NVE by its 32-bit address, which is composed of these three fields, as illustrated in the following figure.

- a 16-bit network address
- an 8-bit node address
- an 8-bit number representing the DDP socket number



Network Number
Node Number
Socket Number
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
254 per node (1 through 2
Dynamically assigned by the node (1 thro
Statically assigned by a bridge on an internet, 0x0000 on a local net

NBP associates an *entity name* (EN) with an NVE. An EN has three components:

- *object,* the name of the object
- *type,* the type of the object
- *zone,* the name of the zone in which the object resides

Each of these three components may be up to 32 characters long; uppercase and lowercase are ignored when comparisons are made. All characters are significant, including trailing spaces. The three components are concatenated to form the entity name. The *object* and *type* fields are separated by a colon (:), and the *zone* field is preceded by an "at" symbol (@). Here is an example of a possible entity name for a LaserWriter called `writer4` in a zone named `doc`:

```
writer4:LaserWriter@doc
```

You can use an equal sign (=) as a wildcard to signify all possible values of the *object* or *type* field. You can use an asterisk (*) in the *zone* field to represent the current zone.

Five functional areas are involved in name binding. A function in the library is provided for each of these functional areas:

- Look up an EN’s internet address.
- Confirm an EN’s internet address.
- Register an EN’s internet address.
- Deregister an EN’s internet address.
- Decompose an EN into its components.

See *Inside AppleTalk* for more information.

NBP library functions

The NBP functions use the following structures.

```
typedef struct at_inet {
     at_net net;
     at_node node;
     at_socket socket;
} at_inet_t;
```

The `at_inet` structure specifies the AppleTalk internet address of a DDP socket end point.

```
typedef struct at_retry {
     short interval;
     short retries;
     u_char backoff;
} at_retry_t;
```

The `at_retry` structure specifies the retry interval and maximum number of retries for a transaction. The *interval* parameter specifies the number of seconds to wait before retrying an NBP request. The *retries* parameter specifies the maximum number of retries for an NBP request. NBP does not use the *backoff* parameter.

```
typedef struct at_nevstr {
     char len;
     char str[NBP_NVE_STR_SIZE];
} at_nvestr_t;
```

The `at_nevstr` structure specifies an NBP entity string. The *len* parameter specifies the length of the string in bytes, and the *str* parameter specifies the character data for the string.

```
typedef struct at_entity {
     at_nvestr_t object;
     at_nvestr_t type;
     at_nvestr_t zone;
} at_entity_t;
```

The `at_entity` structure specifies an entity name. The *object* parameter specifies the name of the object; the *type* parameter specifies the type of the object (for example, Macintosh IIsi); and the *zone* parameter specifies the name of the zone. The strings in the members of this structure are not null-terminated.

```
typedef struct at_nbptuple {
     at_inet_t enu_addr;
     union {
          struct {
          u_char enumerator;
          at_entity_t entity;
     } en_se;
          struct {
          u_char enumerator;
          u_char name[NBP_TUPLE_SIZE];
          } en_sn;
     } en_u;
} at_nbptuple_t;
```

The `at_nbptuple` structure is used to store the name-address pairs that result from querying the NBP name tables by calling `nbp_lookup`(). The strings in the members of this structure are not null-terminated.

All of the NBP functions return –1 to indicate an error and one of the following error codes in `errno`:

EINVAL

Indicates that an EN is invalid.

EADDRNOTAVAIL

Indicates that the requested address is not available.

ETIMEDOUT

Indicates that the request exceeded the maximum number of retries.

The following paragraphs describe the NBP functions.

The `nbp_parse_entity` function

```
int nbp_parse_entity(entity, str);

at_entity_t *entity;

char *str;
```

This function parses *str*, which is a null-terminated string of the form *object*,*object*:*type* or of the form *object*:*type@zone*, and stores the elements in the entity structure specified by *entity*.

The `nbp_make_entity` function

```
int nbp_make_entity(entity, object, type, zone)

at_entity_t *entity;

char *object, *type, *zone;
```

This function stores the value of *object*, *type*, and *zone* (all of which must be null-terminated strings) in the entity structure specified by *entity*. This function returns 0 to indicate success.

The `nbp_confirm` function

```
int nbp_confirm(entity, dest, retry)

at_entity_t *entity;

at_inet_t *dest;

at_retry_t *retry;
```

This function sends a confirmation request to the node specified by *dest* to see if the EN specified by *entity* is still registered at the same address. The object and type members of the entity structure cannot contain wildcards, but the zone member can contain an asterisk (`*`) to represent the current zone. If the EN is still registered, but at a different socket number, the socket member of the `at_inet_t` structure is updated. The *retry* parameter points to an `at_retry_t` structure that specifies the NBP request retry interval in seconds and the maximum retry count. If the value of *retry* is NULL, the retry interval is one second and the maximum retry count is eight. This routine returns 1 to indicate success, 0 to indicate that the EN is not confirmed, and –1 to indicate an error.

The `nbp_lookup` function

```
int nbp_lookup(entity, buf, max, retry)

at_entity_t *entity;

at_nbptuple_t *buf;

int max;

at_retry_t *retry;
```

This function queries the NBP name tables on nodes in the specified AppleTalk zone (specified by the zone member of the entity structure specified by *entity*) and returns a list of NVEs that match the EN specified by *entity*. The value of the zone member of the entity structure can be an asterisk (`*`) to indicate the current zone. The `nbp_lookup`() function stores the list of name/address pairs at the location pointed to by *buf*, which is an array of `at_nbptuple` structures. The parameter *max* specifies the number of `at_nbptuple` structures that *buf* can hold. The parameter *retry* is a pointer to an `at_retry_t` structure, which specifies the retry interval and the maximum number of retries for this NBP transaction.

The `nbp_register` function

```
int nbp_register(entity, fd, retry)
at_entity_t *entity;
int fd;
at_retry_t *retry;
```

This function registers the EN specified by *entity* with the NBP names table on the node. The `nbp_register` function ignores the specified zone name and instead uses the name of the home zone. The value of *fd* is the file descriptor to be registered with the EN, and *retry* is a pointer to an `at_retry_t` structure, which specifies the retry interval and the maximum number of retries for this NBP transaction. If the value of *retry* is NULL, the retry interval is one second and the maximum number of retries is eight.

If an NVE with the same object and type names has already been registered in the current zone, `nbp_register()` rejects the registration by returning –1 and setting `errno` to EADDRNOTAVAIL.

The `nbp_remove` function

```
int nbp_remove(entity, fd)
at_entity_t *entity;
int fd;
```

This function removes the EN specified by *entity* from the NBP name table. None of the members of the entity structure can contain wildcards. The zone member of the entity structure is ignored. The value of *fd* is the file descriptor registered with the EN. Note that only the EN-to-NVE mapping is removed, not the NVE resource itself, so you can still send information to the AppleTalk internet address.

Printer Access Protocol

The Printer Access Protocol (PAP) is designed primarily to communicate with printers. Details of how data is transferred reliably at the PAP level are handled by the protocol itself.

PAP is constructed in a server-client relationship. The server is the printer itself, and the client is the print requester. PAP allows multiple connections and handles connection setup, maintenance, closure and data transfer.

PAP Client library functions

The following sections describe the functions that a PAP client calls.

The `pap_open` function

```
int pap_open(tuple)
```

```
at_nbtuple_t *tuple;
```

This function returns a PAP client file descriptor for a connection with the server whose name and address are specified by *tuple*. You can call `nbp_lookup`() to obtain a valid name and address of a particular PAP server.

The `pap_read` function

```
int pap_read(fd, data, len)
```

```
int fd, len;
```

```
char *data;
```

When called by a PAP client, this function reads data from a client PAP file descriptor that has been opened by calling `pap_open`(). The value of *fd* is a PAP client file descriptor; the value of *data* is a pointer to a buffer into which the data is to be stored; and the value of *len,* which can be no more than 512, specifies the number of bytes to read. When successful, `pap_read`() returns the number of bytes read. When `pap_read`() encounters an end-of-file character, it returns 0.

The `pap_read_ignore` function

```
int pap_read_ignore(fd)
```

```
int fd;
```

This function issues a PAP read request and ignores any returned data. You can use this call to allow printers to function when they want to return status messages.

The `pap_status` function

```
char *pap_status(tuple)
at_nbtuple_t *tuple;
```

This function returns a pointer to a PAP server's status string. The value of *tuple* is a pointer to a structure containing the name and address of a PAP server. You can call `nbp_lookup`() to obtain the value for the *tuple* parameter.

The `pap_write` function

```
int pap_write(fd, data, len, eof, flush)
int fd, len;
int eof, flush;
char *data;
```

This function sends data to a PAP server. The value of *fd* is a PAP client file descriptor. The value of *data* is a pointer to the data being written. The value of *len* , which must be greater than 0, is the amount of data to send. The value of *eof* is TRUE to indicate that no more data will be sent or FALSE to indicate that more data will be sent. *eof* must be set on the last data packet sent. The value of *flush* is TRUE to indicate that all data waiting for PAP writes should be sent (that is, flushed) to the PAP server or FALSE to indicate that a flush is not required. The *flush* parameter is required because PAP runs on top of ATP, which means that PAP writes are queued until a complete ATP response (4 K) is accumulated (that is, *flush* is set to FALSE) or until ATP receives an end-of-message (that is, *flush* is set to TRUE). Setting *eof* to TRUE implies that *flush* is also TRUE. You should set *flush* to TRUE whenever a higher-level protocol, such as a handshake with a LaserWriter, needs to do a write followed by a read.

This function returns a value that is less than 0 to indicate failure.

The `pap_close` function

```
int pap_close(fd)
int fd;
```

This function closes an open PAP client file descriptor. The value of *fd* is the file descriptor to be closed. This routine returns 0 on success; if the file descriptor is no longer open, this routine returns –1.

AppleTalk Data Stream Protocol

The AppleTalk Data Stream Protocol (ADSP) is a symmetric, connection-oriented protocol that guarantees the ordered delivery of full-duplex streams of bytes between two given sockets in an AppleTalk internet. ADSP runs over the Datagram Delivery Protocol DDP and operates at the same layer as AppleTalk Session Protocol (ASP).

Important Unlike the other AppleTalk protocols described previously, there are two interfaces for ADSP: the first is the socket-like interface and the second is the industry standard TLI interface. The following sections describe both interfaces.

ADSP Socket-like Interface

The ADSP socket-like interface provides a programming interface very similar to that of the popular socket interface. For programmers who are familiar with the socket interface used for the Internet protocol suite (TCP/IP), this ADSP interface looks very much the same, except for the names of all ADSP function calls having the ADSP prefix.

Following is the list of the ADSP socket-like function calls.

```
ADSPaccept()

ADSPbind()

ADSPclose()

ADSPconnect()

ADSPfwdreset()

ADSPgetpeername()

ADSPgetsockname()

ADSPgetsockopt()

ADSPlisten()
```

```
ADSPrecv()

ADSPsend()

ADSPsetsockopt()

ADSPsocket()

ASYNCread()

ASYNCread_complete()
```

Some of these function calls require specifying an endpoint address. This address is specified using the structure `at_inet_t` defined as follows:

```
typedef struct at_inet {
  u_char net[2];    /* Network address */
  u_char node; /* Node number */
  u_char socket;    /* Socket number */
} at_inet_t;
```

The following sections provide a detailed description of each ADSP socket-like function call.

The `ADSPaccept` function

```
int ADSPaccept(fd, name, namelen)

int fd;

at_inet_t *name;

int *namelen;
```

This function is used to accept a connection on an ADSP socket. The argument *`fd`* is the socket descriptor of the listening ADSP socket that has been created with the `ADSPsocket`() call. The ADSP socket must have been bound to a listening address via the `ADSPbind`() call and set to listen for connections via the `ADSPlisten`() call.

The argument *`name`* is the result parameter that is filled in with the address of the remote endpoint used in communicating with the connected local endpoint when the connection has been established.

The argument *namelen* is both the calling parameter and result parameter. On calling, this argument must contain a value greater than or equal to the size of the structure `at_inet_t`. On return, it will be filled in with the length of the address of the local endpoint, which in ADSP case is the size of the structure `at_inet_t`.

On success, this function returns the socket descriptor of the connection; otherwise, it returns -1. Also, ADSP protocol can support up to 1024 server connections using this function call.

The `ADSPbind` function

```
int ADSPbind(fd, name, namelen)

int fd;

at_inet_t *name;

int namelen;
```

This function is used to bind an ADSP address to a socket. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the ADSPsocket() call. The ADSP socket must not have already been bound to an ADSP address.

The argument *name* is the parameter that contains a specific socket number which the application wishes the socket to be bound to. If the argument *name* is 0 or the specified socket number (name->socket) is 0, an unused socket number will be assigned by the ADSP protocol module.

The argument *namelen* is the parameter that indicates the length of the address, which in ADSP case is the size of the structure `at_inet_t`.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPclose` function

```
int ADSPclose(fd)

int fd;
```

This function is used to terminate communication on an ADSP socket. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket()` call.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPconnect` function

```
int ADSPconnect(fd, name, namelen)
int fd;
at_inet_t *name;
int namelen;
```

This function is used to establish a connection to a remote ADSP socket. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call. The ADSP socket must have been bound to an ADSP address via the `ADSPbind`() call.

The argument *`name`* is the parameter that contains the address of the remote endpoint to make connection. The address of the remote endpoint could be obtained using the `nbp_lookup`() call which maps the service name of the remote endpoint to an address suitable for use in the `ADSPconnect`() call.

The argument *`namelen`* is the parameter that indicates the length of the address, which in ADSP case is the size of the structure `at_inet_t`.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPfwdreset` function

```
int ADSPfwdreset(fd)
int fd;
```

This function is used to abort the delivery of any outstanding data to the remote endpoint's client. This also causes the two endpoints to be resynchronized. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPgetpeername` function

```
int ADSPgetpeername(fd, name, namelen)
int fd;
```

```
at_inet_t *name;
int *namelen;
```

This function is used to obtain the address of the connected remote endpoint. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *name* is the result parameter that is filled in with the address of the remote endpoint.

The argument *namelen* is both the calling parameter and result parameter. On calling, this argument must contain a value greater than or equal to the size of the structure `at_inet_t`. On return, it will be filled in with the length of the address of the remote endpoint, which in ADSP case is the size of the structure `at_inet_t`.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPgetsocketname` function

```
int ADSPgetsockname(fd, name, namelen)
int fd;
at_inet_t *name;
int *namelen;
```

This function is used to obtain the address of the local endpoint. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *name* is the result parameter that is filled in with the address of the local endpoint.

The argument *namelen* is both the calling parameter and result parameter. On calling, this argument must contain a value greater than or equal to the size of the structure `at_inet_t`. On return, it will be filled in with the length of the address of the remote endpoint, which in ADSP case is the size of the structure `at_inet_t`.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPgetsockopt` function

```
int ADSPgetsockopt(fd, level, optname, optval, optlen)
int fd;
int level;
int optname;
char *optval;
int *optlen;
```

This function is a placeholder that always returns -1 to indicate that the operation is not supported. The `errno` variable is also set to `EOPNOTSUPP`.

```
int ADSPlisten(fd, backlog)
int fd;
int backlog;
```

This function is used to set the local endpoint to the state ready to listen for connections from remote endpoints. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *`backlog`* specifies the maximum length of the queue of pending connections. With the current implementation of the ADSP protocol module, this queue length is always 1 so the *`backlog`* value has no effect.

On success, this function returns 0; otherwise, it returns -1.

The `ADSPrecv` function

```
int ADSPrecv(fd, buf, len, flags)
int fd;
char *buf;
int len;
int flags;
```

This function is used to receive data from a socket. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *buf* is the parameter that contains the address of the buffer to be filled in with the receive data.

The argument *len* is the parameter that specifies the maximum size of the receive buffer.

The argument *flags* is the parameter that specifies either 0 to read the normal data or 1 to read the expedited (attention) data.

On success, this function returns the number of bytes received; otherwise, it returns -1.

The `ADSPsend` function

```
int ADSPsend(fd, buf, len, flags)
int fd;
char *buf;
int len;
int flags;
```

This function is used to send data from a socket to a remote endpoint. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *buf* is the parameter that contains the address of the buffer containing the data to be sent.

The argument *len* is the parameter that specifies the amount of the data to be sent.

The argument *flags* is the parameter that specifies either 0 to write the normal data or 1 to write the expedited (attention) data.

On success, this function returns the number of bytes sent; otherwise, it returns -1.

The `ADSPsetsockopt` function

```
int ADSPsetsockopt(fd, level, optname, optval, optlen)
int fd;
int level;
int optname;
char *optval;
int optlen;
```

This function is used to set options on a socket. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *`level`* should be set to `SOL_SOCKET` (0xffff). The argument *`optname`* is not examined and should be set to 0, since at the present time only one set of options can be set for ADSP. The *`optval`* is the pointer to the structure containing the option values and the *`optlen`* specifies the length of the structure.

To set the local send options, use the `at_adspopt` parameter block, set the *`optval`* pointing to the address of the block and the *`optlen`* to the size of the `at_adspopt` structure.

The `at_adspopt` structure

```
union at_adspopt {
 struct {
  u_short sendBlocking;  /* quantum for data packets */
  u_char sendTimer;      /* send timer in 10-tick
intervals */
  u_char rtmtTimer;      /* for internal use only */
  u_char badSeqMax;      /* threshold for sending
retransmit advice */
  u_char useCheckSum;    /* use ddp packet checksum */
  u_short filler;
  int newPID;                 /* owner's process id */
 } c;
};
```

On success, this function returns 0; otherwise, it returns -1.

Note: If a forked process inherits a socket from its parent process, this function should be used to signify it as the new owner of the socket by setting *newPID* to its process ID.

The `ADSPsocket` function

```
int ADSPsocket(domain, type, protocol)

int fd;

int type;

int protocol;
```

This function is used to create an endpoint (socket) for communication. The arguments *`domain, type`* and *`protocol`* are not examined since this function implicitly knows to create an ADSP socket. However, for future portability, the argument *`domain`* should be `PF_APPLETALK`, the argument *`type`* should be `SOCK_STREAM` and the *`protocol`* parameter should be 0.

On success, this function returns the socket descriptor of the created socket; otherwise, it returns -1.

The `ASYNCread` function and the `ASYNCread_complete` function

The following two special asynchronous functions are provided to support using the `select`() call to wait for incoming ADSP data on a socket. The `select`() call must always be used with these two function calls.

```
int ASYNCread(fd, buf, len)

int fd;

char *buf;

int len;
```

This function is used in conjunction with the `ASYNCread_complete`() function to asynchronously receive data from a socket. This function and the `ASYNCread_complete`() function are designed to support using the `select`() call for reading on a socket descriptor. The argument *`fd`* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket`() call.

The argument *buf* is the parameter that contains the address of the buffer to be filled in with the receive data.

The argument *len* is the parameter that specifies the maximum size of the receive buffer.

On success, this function returns the number of bytes received; otherwise, it returns -1 to indicate failure or 0 to indicate that it has registered a read request to the ADSP protocol module. If this function returns 0, the application can subsequently use the `select()` call to wait for input data.

```
int ASYNCread_complete(fd, buf, len)

int fd;

char *buf;

int len;
```

This function is used in conjunction with the `ASYNCread()` function to asynchronously receive data from a socket. This function and the `ASYNCread()` function are designed to support using the `select()` call for reading on a socket descriptor. The argument *fd* is the socket descriptor of the ADSP socket that has been created with the `ADSPsocket()` call.

The argument *buf* is the parameter that contains the address of the buffer to be filled in with the receive data.

The argument *len* is the parameter that specifies the maximum size of the receive buffer.

On success, this function returns the number of bytes received; otherwise, it returns -1 to indicate failure. This function must only be called if the `ASYNCread()` function has previously been called and had a 0 return value.

The `select()` call is usually used between the `ASYNCread()` function and the `ASYNCread_complete` function to wait for input data, for example:

```
if  ((retlen = ASYNCread(fd, buf, len)) == 0) {

    FD_ZERO(&readset);

    FD_SET(fd, &readset);

    if (select(fd+1, &readset, 0, 0, 0) == 1) {

        retlen = ASYNCread_complete(fd, buf, len);
        }
    }
```

Important ADSP incoming normal data can be handled with the UNIX signal mechanism. Incoming normal data is signaled through SIGIO. The user routine must register a handler for normal data using the `signal`() call. The alternative to this approach is to use the `select`() call, in conjunction with the `ASYNCread`() and `ASYNCread_complete`() functions described above.

ADSP Attention data is handled with the UNIX signal mechanism. Incoming attention data is signaled through SIGURG. Like the signal call for SIGIO used for normal data, the user routine must register a handler for attention data using the `signal`() call.

ADSP TLI Interface

The ADSP TLI interface provides the standard programming interface using the Transport Layer Interface (TLI) standard. For programmers who are familiar with the TLI interface used for the Internet protocol suite (TCP/IP), this ADSP interface is easy to use. The following list shows the TLI calls supported by the ADSP TLI interface:

```
t_accept()
t_alloc()
t_bind()
t_close()
t_connect()
t_error()
t_free()
t_getinfo()
t_getstate()
t_listen()
t_look()
```

```
t_optmgmt()
t_open()
t_rcv()
t_snd()
t_sync()
t_unbind()
```

When programming using this interface, refer to the AIX system manual (or any UNIX application programming interface manual) for description of the TLI interface in general. The only thing that is specific to the ADSP protocol is the specification of an endpoint address for some of the above TLI function calls. This address is specified using the structure `at_inet_t` defined as follows:

```
typedef struct at_inet {
  u_char net[2];    /* Network address */
  u_char node;      /* Node number */
  u_char socket;    /* Socket number */
} at_inet_t;
```

Also, the device name to be specified in the `t_open`() call for ADSP should be `dev/adsp`.

Important To abort the delivery of any outstanding data to the remote endpoint's client, use the `ADSPfwdreset`() call described in the ADSP Socket-like Interface section.

Note: This interface is a bit slower than that of the ADSP socket-like interface due to the specific implementation of the AIX TLI library and the semantics of the TLI interface itself. The performance difference is most noticeable when receiving data using the `t_rcv`() function as compared to the `ADSPrecv`() function.

AppleTalk Session Protocol

AppleTalk Session Protocol (ASP) is designed primarily to support the AppleTalk Filing Protocol (AFP). However, it can be used by any application desiring a session-type of communication.

ASP is constructed in a server-client relationship. The server is the service provider (such as an AFP server) and the client is the service user (such as an AFP client). ASP allows multiple concurrent connections—one for each session, and handles session set up, maintenance, closure, and data transfer.

In the following description of the ASP library functions, the ASP client refers to either the server or workstation application that uses the ASP library functions, the server client refers to the server application (such as an AFP server module) and the workstation client refers to the workstation application (such as an AFP client module).

ASP library functions

The AppleTalk API provides the following ASP function calls:

```
SPAttention()
SPCloseSession()
SPCmdReply()
SPCommand()
SPConfigure()
SPEnableSelect()
SPGetParms()
SPGetRemEntity()
SPGetReply()
SPGetRequest()
SPGetSession()
SPGetStatus()
SPInit()
SPLook()
```

```
SPNewStatus()

SPOpenSession()

SPRegister()

SPRemove()

SPSetPid()

SPWrite()

SPWrtContinue()

SPWrtReply()
```

Some of the above function calls require specifying an endpoint address. This address is specified using the structure `at_inet_t` defined as follows:

```
typedef struct at_inet {
  u_char net[2];    /* Network address */
  u_char node; /* Node number */
  u_char socket;    /* Socket number */
} at_inet_t;
```

The following sections contain detailed descriptions of each ASP function call. `OSErr` is defined as type `int`.

The `SPAttention` function

```
int SPAttention(SessRefNum, AttentionCode, SPError,
NoWait)

int SessRefNum;

unsigned short AttentionCode;

OSErr *SPError;

int NoWait;
```

This function is used by the server client to send the attention code to the workstation client and waits for an acknowledgment. The argument *`SessRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() call.

The argument `AttentionCode` is the 2-byte attention code to be delivered to the workstation client.

The argument `SPError` is the result parameter that is filled in with the error code when the function call fails.

The argument `NoWait`, if set to non-zero, indicates the desire not to wait for the reply data after the attention is sent. The workstation client must later receive the reply data using the `SPGetReply`() call. Moreover, the server client can use the `select`() call to probe the availability of the reply data.

On success, this function returns 0; otherwise, it returns -1.

The SPCloseSession function

```
int SPCloseSession(SessRefNum, SPError)
int SessRefNum;
OSErr *SPError;
```

This function is used by the ASP client to close an ASP session. The argument `SessRefNum` is the socket descriptor of the ASP socket that has been created with the `SPOpenSession`() or `SPGetSession`() call.

The argument `SPError` is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

Note: When multiple processes share the same session socket, only the last one that actually closes the session socket should call this function. If a process is not the last one to close the session socket, it must call the regular AIX `close`(*SessRefNum*) system call. For instance, when a server process accepts a connection, spawns off a child process to handle the connection and then is no longer interested in the connection socket, it must call the `close`(*SessRefNum*) to remove reference to the socket descriptor.

The SPCmdReply function

```
int SPCmdReply(SessRefNum, ReqRefNum, CmdResult,
CmdReplyData,  CmdReplyDataSize,SPError)
int SessRefNum;
```

```
unsigned short ReqRefNum;
int CmdResult;
char *CmdReplyData;
int CmdReplyDataSize;
OSErr *SPError;
```

This function is used by the server client to respond to the Command request from the workstation client. The argument *`SessRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() call.

The argument *`ReqRefNum`* is the same parameter as that returned by the corresponding `SPGetRequest`() call.

The argument *`CmdResult`* is 4-byte command result to be sent back to the workstation client.

The argument *`CmdReplyData`* is the pointer to the buffer containing the reply data and the argument *`CmdReplyDataSize`* specifies the size of the data.

The argument *`SPError`* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPCommand` function

```
int SPCommand(SessRefNum, CmdBlock, CmdBlockSize,
ReplyBuffer,   ReplyBufferSize, CmdResult,
ActRcvdReplyLen, SPError, NoWait)
int SessRefNum;
char *CmdBlock;
int CmdBlockSize;
char *ReplyBuffer;
int ReplyBufferSize;
int *CmdResult;
int *ActRcvdReplyLen;
```

```
OSErr *SPError;
int NoWait;
```

This function is used by the workstation client to send a command to the server client, and supports only one command at a time in progress; for example, if you use the *`NoWait`* feature to attempt to send multiple commands at a time it will not work. The argument *`SessRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPOpenSession`() call.

The argument *`CmdBlock`* is the pointer to the buffer containing the command data and the argument *`CmdBlockSize`* specifies the size of the data.

The argument *`ReplyBuffer`* is the pointer to the buffer to be filled in with any reply data and the argument *`ReplyBufferSize`* specifies the size of the buffer.

The argument *`CmdResult`* is the result parameter that is filled in with the 4-byte command result to be received from the server client.

The argument *`ActRcvdReplyLen`* is the result parameter that indicates the actual length of the reply data stored in *`ReplyBuffer.`*

The argument *`SPError`* is the result parameter that is filled in with the error code when the function call fails.

The argument *`NoWait`*, if set to non-zero, indicates the desire not to wait for the reply data after the command is sent. The workstation client must later receive the reply data using the `SPGetReply`() call. Moreover, the workstation client can use the `select`() call, only if `SPEnableSelect`() has previously been called to enable the select option, to probe the availability of the reply data.

On success, this function returns 0; otherwise, it returns -1.

The `SPConfigure` function

```
void SPConfigure(TickleInterval, SessionTimer, Retry)
int *TickleInterval;
int *SessionTimer;
```

```
at_retry_t *Retry;
```

This function is used to change the system default configuration for ASP. The argument *TickleInterval* specifies in seconds the periodic interval that tickles are sent between a server client and workstation client; the argument *SessionTimer* specifies in seconds the session maintenance timeout; and the argument *Retry* specifies the retries used by ASP client for session opening and getting status information. The *TickleInterval* default value is 30 seconds; the *SessionTimer* default value is 120 seconds; and the *Retry* default value is 10 retries of 1 second interval.

If a value of zero is specified for any of the arguments of this function call, the corresponding default value is used.

The `SPEnableSelect` function

```
int SPEnableSelect(SessRefNum, SPError)
int SessRefNum;
OSErr *SPError;
```

This function must be used to enable using the `select()` call for no-wait operations. Although this function is available for easy of programming when an application must wait for multiple events on multiple descriptors, it is highly recommended that it is not used at all possible if there is an alternate mechanism since using this option will cause noticeable degradation in performance.

The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession()` or `SPOpenSession()` call.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPGetParms` function

```
void SPGetParms(MaxCmdSize, QuantumSize)
int *MaxCmdSize;
int *QuantumSize;
```

This function is used by the ASP client to retrieve the maximum value of the command block size and the quantum size.

The argument *`MaxCmdSize`* is the result parameter that is filled in with the value of the maximum size of a command block.

The argument *`QuantumeSize`* is the result parameter that is filled in with the value of the maximum size for a command reply or a write.

The `SPGetRemEntity` function

int SPGetRemEntity(*SessRefNum*, *SessRemEntityIdentifier*, *SPError*)

int *SessRefNum*;

at_inet_t **SessRemEntityIdentifier*;

OSErr **SPError*;

This function is used to obtain the address of the remote endpoint. The argument *`SessRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() or `SPOpenSession`() call.

The argument *`SessRemEntityIdentifier`* is the result parameter that is filled in with the address of the remote endpoint.

On success, this function returns 0; otherwise, it returns -1.

The `SPGetReply` function

int SPGetReply(*SessRefNum*, *ReplyBuffer*,
     ReplyBufferSize, *CmdResult*, *ActRcvdReplyLen*, *SPError*)

int *SessRefNum*;

char **ReplyBuffer*;

int *ReplyBufferSize*;

int **CmdResult*;

int **ActRcvdReplyLen*;

OSErr **SPError*;

This function is used by the ASP client to receive the reply for a previous request. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession()` or `SPOpenSession()` call.

The argument *ReplyBuffer* is the pointer to the reply buffer that is filled in with the reply data and the argument *ReplyBufferSize* specifies the size of the reply buffer.

The argument *CmdResult* is the result parameter that is filled in with the 4-byte command result.

The argument *ActRcvdReplyLen* is the result parameter that indicates the actual length of the reply data stored in *ReplyBuffer.*

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; It returns -1 for an error condition. However, if the message happens to be a request instead of a reply, this function returns the request type and *CmdResult* is filled in with the request identifier.

The `SPGetRequest` function

```
int SPGetRequest(SessRefNum, ReqBuffer,

     ReqBufferSize, ReqRefNum, ReqType, ActRcvdReqLen,
SPError)

int SessRefNum;

char *ReqBuffer;

int ReqBufferSize;

int *ReqRefNum;

int *ReqType;

int *ActRcvdReqLen;

OSErr *SPError;
```

This function is used by the ASP client to receive a request or write-continue. The argument *`SessRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() or `SPOpenSession`() call.

The argument *`ReqBuffer`* is the pointer to the request buffer that is filled in with the request data and the argument *`ReqBufferSize`* specifies the size of the request buffer.

The argument *`ReqRefNum`* is the result parameter that is filled in with the request identifier.

The argument *`ReqType`* is the result parameter that is filled in with the ASP-level request type.

The argument *`ActRcvdReqLen`* is the result parameter that indicates the actual length of the request data stored in *`ReqBuffer.`*

The argument *`SPError`* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; It returns -1 for error condition. However, if the message happens to be a reply, instead of a request, for some previous asynchronous request, this function returns 1 and *`ReqRefNum`* is filled in with the command result.

The `SPGetSession` function

```
int SPGetSession(SLSRefNum, SessRefNum,SPError)

int SLSRefNum;

int *SessRefNum;

OSErr *SPError;
```

This function is used to accept a connection on an ASP socket. The argument *`SLSRefNum`* is the socket descriptor of the ASP socket that has been created with the `SPInit`() call.

The argument *`SessRefNum`* is the result parameter that is filled in with the socket descriptor used in communicating with the connected remote endpoint when the connection has been established.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1. Also, ASP protocol can support 2048 connections using this function call.

The `SPGetStatus` function

```
int SPGetStatus(SLSEntityIdentifier, StatusBuffer,
     StatusBufferSize, ActRcvdStatusLen, SPError)
at_inet_t *SLSEntityIdentifier;
char *StatusBuffer;
int StatusBufferSize;
int *ActRcvdStatusLen;
OSErr *SPError;
```

This function is used by the workstation client to obtain the status information block of an ASP service provider. The argument *SLSEntityIdentifier* is the address of the service provider to be queried.

The argument *StatusBuffer* is the pointer to the status buffer that is filled in with the status data and the argument *StatusBufferSize* specifies the size of the status buffer.

The argument *ActRcvdStatusLen* is the result parameter that indicates the actual length of the status data stored in *StatusBuffer.*

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPInit` function

```
int SPInit(SLSEntityIdentifier, ServiceStatusBlock,
     ServiceStatusBlockSize, SLSRefNum, SPError)
at_inet_t *SLSEntityIdentifier;
char *ServiceStatusBlock;
```

```
int ServiceStatusBlockSize;
int *SLSRefNum;
OSErr *SPError;
```

This function is used by a Server service provider to pass its network-dependent service identifier as well as its service status block to ASP. The argument *SLSEntityIdentifier* is the address of the service provider which can be obtained using the `SPRegister`() call.

The argument *ServiceStatusBlock* is the pointer to the status buffer that contains the status data and the argument *ServiceStatusBlockSize* specifies the size of the status buffer.

The argument *SLSRefNum* is the result parameter that is filled in with the socket descriptor for the service.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPLook` function

```
int SPLook(SessRefNum, SPError)
int SessRefNum;
OSErr *SPError;
```

This function is used by the ASP client to determine whether the message available at the local end-point is a request or a reply. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() or `SPOpenSession`() call.

On success, this function returns 0 indicating a reply or 1 indicating a request; otherwise, it returns -1 for error or no message.

The `SPNewStatus` function

```
int SPNewStatus(SLSRefNum, ServiceStatusBlock,
     ServiceStatusBlockSize, SPError)
int SLSRefNum;
```

```
char *ServiceStatusBlock;
int ServiceStatusBlockSize;
OSErr *SPError;
```

This function is used by a Server service provider to update its service status block. The argument *SLSRefNum* is the socket descriptor of the service provider that has been created with the `SPInit`() call.

The argument *ServiceStatusBlock* is the pointer to the status buffer that contains the status data and the argument *ServiceStatusBlockSize* specifies the size of the status buffer.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPOpenSession` function

```
int SPOpenSession(SLSEntityIdentifier, AttentionCode,
     SessRefNum,SPError)
at_inet_t *SLSEntityIdentifier;
void (*AttentionCode)();
int *SessRefNum;
OSErr *SPError;
```

This function is used to establish a connection to a Server service provider. The argument *SLSEntityIdentifier* is the address of the service provider.

The argument *AttentionCode* is the address of the routine to call when an attention message is received. For AIX, this feature is not supported so this argument is ignored.

The argument *SessRefNum* is the result parameter that is filled in with the socket descriptor used in communicating with the connected remote endpoint when the connection has been established.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPRegister` function

`int SPRegister(`*`SLSEntity`*`,` *`Retry`*`,` *`SLSEntityIdentifier`*`,` *`SPError`*`)`

`at_entity_t *`*`SLSEntity`*`;`

`at_retry_t *`*`Retry`*`;`

`at_inet_t *`*`SLSEntityIdentifier`*`;`

`OSErr *`*`SPError`*`;`

This function is used to register a Server service provider specified by *SLSEntity* with the NBP names table on the node. The *Retry* is a pointer to an `at_retry_t` structure, which specifies the retry interval and the maximum number of retries for this NBP transaction. If the value of *Retry* is 0, the retry interval is one second and the maximum number of retries is eight.

The argument *`SLSEntityIdentifier`* is the result parameter that is filled in with the assigned address for the service provider.

The argument *`SPError`* is the result parameter that is filled in with the error code if the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The `SPRemove` function

`int SPRemove(`*`SLSEntity`*`,` *`SLSEntityIdentifier`*`,` *`SPError`*`)`

`at_entity_t *`*`SLSEntity`*`;`

`at_inet_t *`*`SLSEntityIdentifier`*`;`

`OSErr *`*`SPError`*`;`

This function is used to remove the Server service provider specified by *SLSEntity* from the NBP names table. None of the members of the entity structure can contain wildcards. The zone member of the entity structure is ignored. The value of *SLSEntityIdentifier* is the address of the service provider obtained from the `SPRegister`() call.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The SPSetPid function

```
int SPSetPid(SessRefNum, SessPid, SPError)
int SessRefNum;
int SessPid;
OSErr *SPError;
```

This function should be used by a forked process to specify it as the owner of the socket descriptor. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession()` or `SPOpenSession()` call.

The argument *SessPid* is process id of the calling process.

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

The SPWrite function

```
int SPWrite(SessRefNum, CmdBlock, CmdBlockSize, WriteData,
     WriteDataSize, ReplyBuffer,   ReplyBufferSize,
CmdResult,
     ActLenWritten, ActRcvdReplyLen, SPError, NoWait)
int SessRefNum;
char *CmdBlock;
int CmdBlockSize;
char *WriteData;
int WriteDataSize;
char *ReplyBuffer;
int ReplyBufferSize;
int *CmdResult;
int *ActLenWritten;
int *ActRcvdReplyLen;
OSErr *SPError;
int NoWait;
```

This function is used by the workstation client to send a write request to the server client. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPOpenSession`() call.

The argument *CmdBlock* is the pointer to the buffer containing the command data and the argument *CmdBlockSize* specifies the size of the data.

The argument *WriteData* is the pointer to the buffer containing the data to be written to the server client and the argument *WriteDataSize* specifies the size of the data.

The argument *ReplyBuffer* is the pointer to the buffer to be filled in with any reply data and the argument *ReplyBufferSize* specifies the size of the buffer.

The argument *CmdResult* is the result parameter that is filled in with the 4-byte command result to be received from the server client.

The argument *ActLenWritten* is the result parameter that indicates the actual amount of the data written to the server client*.*

The argument *ActRcvdReplyLen* is the result parameter that indicates the actual length of the reply data stored in *ReplyBuffer.*

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

The argument *NoWait*, if set to non-zero, indicates the desire not to wait for the write-continue request from the server client after the write request is sent. The workstation client must later receive the write-continue request using the `SPGetRequest`() call, send the write data using the `SPCmdReply`() call and then receive the reply data using the `SPGetReply`() call. Moreover, the workstation client can use the `select`() call, only if `SPEnableSelect`() has previously been called to enable the select option, to probe the availability of the write-continue request and the reply data from the server client.

On success, this function returns 0; otherwise, it returns -1.

The `SPWrtContinue` function

```
int SPWrtContinue(SessRefNum, ReqRefNum, Buff, BuffSize,
ActLenRcvd, SPError, NoWait)

int SessRefNum;

int ReqRefNum;

char *Buff;

int BuffSize;

int *ActLenRcvd;

OSErr *SPError;

int NoWait;
```

This function is used by the server client, in response to the write request from the workstation client, to notify the workstation client to continue with writing the data. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() call.

The argument *ReqRefNum* is the same parameter as that returned by the corresponding `SPGetRequest`() call.

The argument *Buff* is the pointer to the receive buffer that is filled in with the write data and the argument *BuffSize* specifies the size of the receive buffer.

The argument *ActLenRcvd* is the result parameter that indicates the actual length of the write data stored in *Buff.*

The argument *SPError* is the result parameter that is filled in with the error code when the function call fails.

The argument *NoWait*, if set to non-zero, indicates the desire not to wait for the write data from the workstation client after the write-continue request is sent. The server client must later receive the write data using the SPGetReply() call. Moreover, the server client can use the `select`() call, only if `SPEnableSelect`() has previously been called to enable the select option, to probe the availability of the write data from the workstation client.

On success, this function returns 0; otherwise, it returns -1.

The `SPWrtReply` function

```
int SPWrtReply(SessRefNum, ReqRefNum, CmdResult,
CmdReplyData,
     CmdReplyDataSize, SPError)
int SessRefNum;
int ReqRefNum;
int CmdResult;
char *CmdReplyData;
int CmdReplyDataSize;
OSErr *SPError;
```

This function is used by the server client to terminate, either successfully or unsuccessfully, the write request from the workstation client. The argument *SessRefNum* is the socket descriptor of the ASP socket that has been created with the `SPGetSession`() call.

The argument *ReqRefNum* is the same parameter as that returned by the corresponding SPGetRequest() call.

The argument *CmdResult* is 4-byte command result to be sent back to the workstation client.

The argument *CmdReplyData* is the pointer to the buffer containing the reply data and the argument *CmdReplyDataSize* specifies the size of the data.

The argument `SPError` is the result parameter that is filled in with the error code when the function call fails.

On success, this function returns 0; otherwise, it returns -1.

Section IV Manual Pages for the Network Server

This section contains the following man pages:

Command Reference—Section 1

`atlookup`

`atprint`

`at_cho_prn`

`atstatus`

`eject`

System Administrator's Reference—Section 1M

`appleping`

`atconfig`

`cmdshld`

`discusd`

`javelind`

`mandeld`

`ppcd`

`ppcd.conf`

Programmer's Reference—Section 3

`OF_child`

`OF_getprop`

`OF_hdl2path`

`OF_nextprop`

`OF_parent`

`OF_path2hdl`

`OF_peer`

`pci_cfgrw`

`resolve_gc_offset`

`resolve_intr_level`

```
resolve_pci_config_space
resolve_pci_io_space
resolve_pci_mem_space
```

NAME

`atlookup`—looks up network-visible entities (NVEs) registered on the AppleTalk network system

SYNOPSIS

`atlookup` [`-d`] [`-r` *nn*] [`-s` *ss*] [`-x`] [*object*[`:`*type*[@*zone*]]]

`atlookup -z` [`-C`]

ARGUMENTS

`-C` Prints zones in multiple columns.

`-d` Prints the network address in decimal numbers.

object

Specifies the name of the NVE to be looked up.

`-r` *nn*

Specifies the number of retries the system employs if the first lookup is unsuccessful. The default number of retries is 8.

`-s` *ss*

Instructs `atlookup` to wait a certain number (*ss*) of seconds between consecutive attempts to complete a lookup successfully. As a default, retries are spaced one second apart.

type

Specifies the type of the NVE to be looked up.

`-x` Prints the 8-bit ASCII characters on output as hexadecimal numbers of the form *XX* (where *X* is a hexadecimal digit). This option is useful when you are using a terminal other than the system console.

`-z` Lists all zones in the network.

zone

Specifies the zone in which the lookup is to be performed. You can use an asterisk (`*`) instead of a zone name to indicate the current zone. If you don't specify a zone name, the current zone is the default.

The *object* and *type* arguments can contain wildcard characters. The equal sign (=) indicates a wildcard lookup. For wildcard lookups to work correctly with all nodes, the wildcard character must be the only character specified in the string. However, AppleTalk

Phase 2 nodes also honor a single embedded wildcard character (=). Under this scheme, one wildcard character can appear anywhere in the string and can match 0 or more characters. Note, however, that although an embedded = is acceptable in *object* and *type* arguments of `atlookup`, only the nodes implementing AppleTalk Phase 2 protocols respond to such a query. For this reason, the resulting list of NVEs may be incomplete.

DESCRIPTION

`atlookup` uses the Name Binding Protocol (NBP) to look up names and addresses of the specified NVEs.

The default is to look up all the entities (of all types) in the current zone. Specifying the object, type, or zone on the command line changes the scope of the lookup.

Information about the NVEs is displayed in a table, one NVE per line. Each line gives the name of the object, its type, and its zone and the numbers of the network, node, and socket.

EXAMPLES

This command looks up all NVEs registered in the local AppleTalk zone:

```
atlookup
```

In response, the system displays output similar to this:

```
Found 5 entries in zone My-Zone

6b5b.c3.ea 3-Eyed Monster:LaserWriter

6b5b.80.fd 3-Eyed Monster Spooler:LaserWriter

6b14.84.ea Incognito:LaserWriter

6b19.a3.fd Light of Day:AFPServer

6b51.27.fd Nets-R-Us Spooler:LaserWriter
```

In an extended AppleTalk network, this command displays all NVEs (of any type) in the current zone whose names start with *L* and end in *y*:

```
atlookup L=y:=
```

The output might be similar to this:

```
Found 1 entries in zone My-Zone

6b19.a3.fd Light of Day:AFPServer
```

FILES

`/usr/bin/atlookup`

Executable file

SEE ALSO

at_cho_prn(1), atprint(1), atstatus(1)

Inside AppleTalk

NAME

`atprint`—transfers data to a printer by using AppleTalk protocols

SYNOPSIS

`atprint` [*printer-name*[`:`*printer-type*[@*zone*]]]

ARGUMENTS

printer-name

Specifies the name of the printer you want to use.

printer-type

Specifies the type of printer, such as `LaserWriter` or `ImageWriter`. Use this argument when you want to allow the network to select the printer, but only a printer of a given type. If you omit this argument, `LaserWriter` is the printer type used by default.

For example, when the printer name is specified with wildcards, the printer used is the one chosen by the network. (See `atlookup`(1) for an explanation of wildcards.) By supplying `LaserWriter` as the printer type in a case such as this, you can restrict the network to choosing a printer that can handle PostScript instructions.

The full range of possible replacement values for *printer-type* depends on the configuration of your network. Each different type of printer broadcasts its type and name when it registers itself with the network. You can use `atlookup` to obtain a report showing this information for all the AppleTalk devices on your network (see `atlookup`(1)).

zone

Specifies the AppleTalk zone in which the printer resides. If you omit this argument or specify it as an asterisk (`*`), the local zone is used.

DESCRIPTION

`atprint` uses a printing protocol to establish a connection to an AppleTalk printer, where it sends data received on its standard input until it reaches an end-of-file character. When it detects an end-of-file character, `atprint` closes the AppleTalk session with the printer, allowing other users to gain access to the printer.

You can select the destination AppleTalk printer through the command-line arguments as described earlier in this manual page. If you do not specify any of these arguments, `atprint` uses the printer that was last selected with `at_cho_prn` (see `at_cho_prn`(1)).

Often the printer you access by way of an AppleTalk connection is a LaserWriter printer. Many LaserWriter models are PostScript printers. If you are using such a LaserWriter, the data that you send it must already be translated into the PostScript page-description language. For example, the `psdit` command translates the output from `troff` (invoked with the `-Tpsc` option) into PostScript:

```
    troff -Tpsc -mm file | psdit | atprint
```

The `atprint` command displays one or more messages indicating the AppleTalk printer with which it is communicating and possibly many printer status messages (for example, a message indicating that another print job is occupying the printer).

Note that `atprint` does not honor requests from a LaserWriter regarding the downloading of fonts. Likewise, it does not add a PostScript header to the beginning of the data stream in the same manner as the printer drivers in the Macintosh operating system. However, the command line in the preceding example does provide a PostScript header because `psdit` adds its own header as part of the PostScript conversion process.

In AppleTalk programming terms, the arguments make up a network-visible entity (NVE), where

printer-name[`:`*printer-type*[`@`*zone*]]

corresponds to the AppleTalk object, type, and zone:

object`:`*type*`@`*zone*

EXAMPLES

This command line converts a plain text file into PostScript and then submits it to `joe's printer`:

```
    enscript -p -file|atprint "joe's printer"
```

The `psdit` command translates the output from `troff` (invoked with the `-Tpsc` option) into PostScript:

```
    troff -Tpsc -mm file | psdit | atprint
```

WARNINGS

The `atprint` command does not process the input files as does `lpr`. To print ASCII files properly on a PostScript printer with `atprint`, you must preprocess the files with `pstext` or `enscript`. Likewise, you must preprocess files produced by `troff` with `psdit`.

FILES

`/usr/bin/atprint`

Executable file

SEE ALSO

at_cho_prn(1), atlookup(1), atstatus(1), enscript(1), lpr(1), psdit(1), pstext(1)

AppleTalk chapters in the *Using AIX, AppleTalk Services, and Macintosh Utilities with the Network Server*

NAME

at_cho_prn—allows you to choose a default printer on the AppleTalk network

SYNOPSIS

at_cho_prn [*type*[@*zone*]]

ARGUMENTS

type[@*zone*]

Specifies the type of printer to be used, and the area (*zone*) in which it resides. If you don't use the *type* argument on the command line, at_cho_prn displays all entities of the types LaserWriter and ImageWriter. The system prompts you to select a printer by entering the appropriate number from the printer list. If you don't enter the @*zone* part of the argument on the command line, at_cho_prn lists all the zones in the internet and prompts you to choose the zone in which you want to select your default printer.

DESCRIPTION

at_cho_prn displays a list of printer selections and saves the name of at_cho_prn command checks the network to determine which printers are registered on that network.

After you specify the zone, at_cho_prn lists the printers (of the specified type) available in that zone.

EXAMPLES

The command

```
at_cho_prn 'LaserWriter@*'
```

produces output similar to this:

```
ITEM  NET-ADDR    OBJECT : TYPE
  1: 56bf.af.fc AnnLW:LaserWriter
  2: 56bf.ac.cc TimLW:LaserWriter


ITEM number (0 to make no selection)?
```

where NET-ADDR is the AppleTalk internet address (printed in hexadecimal numbers) of the printer's listener socket, and OBJECT:TYPE is the name of the registered printer and its type.

FILES

/usr/bin/at_cho_prn

Executable file

SEE ALSO

atlookup(1), atprint(1), atstatus(1)

Inside AppleTalk

AppleTalk chapters in *Using AIX, AppleTalk Services, and Mac OS Utilities on the Network Server*

NAME

`atstatus`—displays status information from an AppleTalk device

SYNOPSIS

`atstatus` [*object* [`:`*type* [@*zone*]]]

ARGUMENTS

object

Specifies the name of the AppleTalk device. Wildcard characters are not permitted. If you don't specify the AppleTalk device, `atstatus` uses the system default. If the name contains spaces, put quotation marks around the name, as in this example:

```
atstatus "Bill's Print Shop"
```

type

Specifies the type of device. If you don't specify the *type* argument, the default is `LaserWriter`. You must supply a *type* argument if you supply a *zone* argument.

zone

Specifies the zone in which the AppleTalk device resides. If you don't specify the zone, the system defaults to `*`, your local zone.

DESCRIPTION

`atstatus` gets the status string from an AppleTalk device, such as a LaserWriter printer.

FILES

`/usr/bin/atstatus`

Executable file

SEE ALSO

`at_cho_prn`(1), `atlookup`(1), `atprint`(1)

Inside AppleTalk

NAME

`eject`—ejects a 3.5-inch floppy disk from the disk drive

SYNOPSIS

`eject` [*floppy-node*]

ARGUMENTS

floppy-node

Specifies the type of floppy driver to eject the disk. If you do not specify the type of floppy driver, the command determines the type for you and ejects the disk.

DESCRIPTION

`eject` causes a 3.5-inch disk drive (see `fd`(7)) to eject an inserted disk.

FILES

`/bin/eject`

Executable file

SEE ALSO

`fd`(7) in the AIX 4.1 manual pages.

NAME

`appleping`—tests the AppleTalk network by sending packets to a named host to obtain a response

SYNOPSIS

`appleping` *net-node* [*packet-size* [*npackets*]]

`appleping` *name*:*type*[@*zone*][*packet-size* [*npackets*]]

ARGUMENTS

name:*type*[@*zone*]

Specifies the host computer by its name, type, and zone. If you do not specify a value for *zone*, `appleping` uses the local zone.

net-node

Specifies the host by its network node number. To see the network node number for a particular system, use the `atlookup` command.

npackets

Specifies how many packets to send before terminating.

packet-size

Specifies the size of each packet in bytes. The value of *packet-size* must be between 14 and 599. The default packet size is 64 bytes.

DESCRIPTION

`appleping` repeatedly sends AppleTalk Echo Protocol request packets to the specified computer and reports whether a reply was received. The `appleping` command continues to send packets and to display the result until the number of packets specified by *npackets* is reached or until you issue the interrupt character (usually by pressing CONTROL-C). Before exiting, `appleping` displays a summary of statistics.

FILES

`/etc/appleping`

Executable file

SEE ALSO

`appletalk`(1M)

`atlookup`(1)

NAME

`atconfig`—allows you to configure and display AppleTalk network interfaces and status

SYNOPSIS

`atconfig -s` [*interface*] [`boot|now|both`]

`atconfig -m` [*interface*] [*packets*] [*config-file*] [*route-table-entries*] [*zone-table-entries*] [`boot|now|both`]

`atconfig` [`-u`] [`-q`] [`-p`]

OPTIONS

`-m` [*packets*] [*config-file*] [*route-table-entries*] [*zone-table-entries*] [`boot|now|both`]

Starts AppleTalk in multi-port mode. The *packets* parameter specifies the maximum number of packets per second that the AppleTalk stack can route through its network interfaces. The *config-file* parameter specifies the pathname of the configuration file, such as `/etc/appletalk.cfg`.

The *route-table-entries* parameter specifies the maximum number of route table entries that the AppleTalk stack handles. The *zone-table-entries* parameter specifies thntries that the AppleTalk stack handles. You must tune both of these entries to accommodate the your network environment. If the number you select is too small, AppleTalk will not be able to handle all the traffic; if the number is too large, system memory will be used inefficiently and system performance could be degraded.

Optionally, you can specify when to start AppleTalk—at system restart, now, or both.

`-p` Displays the AppleTalk interface (EtherTalk), zone name, network number, and node ID, as stored in PRAM.

`-q` Query the status of AppleTalk.

`-s` [*interface*] [`boot|now|both`]

Starts AppleTalk in single-port mode. The *interface* parameter identifies the hardware interface that AppleTalk is to use, such as `et0` or `et1`. The default hardware interface is `et0`. Optionally, you can specify when to start AppleTalk—at system restart, now, or both.

`-u` Stop AppleTalk.

DESCRIPTION

`atconfig` displays the status of an active AppleTalk interface, selects an AppleTalk interface, and makes AppleTalk active or inactive. Any user can display statistics, but only users logged in as `root` can select an AppleTalk interface or make it active or inactive.

EXAMPLES

To start AppleTalk in single port mode with the built-in Ethernet interface, use this command:

```
atconfig -s et0
```

To start AppleTalk in multi-port mode with a maximum number of routed packets of 3000, the configuration file `/etc/appletalk.cfg`, a maximum of 512 route entries and a maximum of 256 zone entries, enter this command:

```
atconfig -m 3000 /etc/appletalk.cfg 512 256
```

FILES

`/usr/bin/atconfig`

Executable file

SEE ALSO

AppleTalk chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

`cmdshld`—allows Mac OS clients to access a Network Server through windowed VT100-compatible terminal emulators

SYNOPSIS

`ppcd`

DESCRIPTION

`cmdshld` runs the CommandShell daemon, which allows Mac OS clients to access a Network Server through a multi-windowed VT100-compatible terminal emulator.

FILES

`/etc/cmdshld`

Executable file

SEE ALSO

Section 1, “Developing Mac OS Clients for the Network Server”

AppleTalk services chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

`discusd`—the Network Server daemon for the Disk Management Utility

SYNOPSIS

```
discusd
```

DESCRIPTION

`discusd` runs the Disk Management Utility daemon (the Network Server component of the Disk Management Utility). The Disk Management Utility application allows you to manage logical volumes on an Network Server from a remote Mac OS computer.

FILES

`/etc/discusd`

Executable file

SEE ALSO

Section 1, “Developing Mac OS Clients for the Network Server”

AppleTalk services chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

`javelind`—allows Mac OS clients to connect to a Network Server and obtain system information

SYNOPSIS

`javelind`

DESCRIPTION

`javelind` runs the daemon for the Status Demo AppleTalk Services application (the Network Server component of the Javelin Macintosh utility), which allows Macintosh clients to obtain and display system information from a Network Server.

FILES

`/etc/javelind`

Executable file

SEE ALSO

Section 1, “Developing Mac OS Clients for the Network Server”

AppleTalk services chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

`mandeld`—allows Mac OS clients to connect to a Network Server to create Mandel fractals

SYNOPSIS

`mandeld`

DESCRIPTION

`mandeld` runs the Mandel daemon, (the Network Server component of the Mandel Macintosh utility), which allows Macintosh clients to use Network Servers to create Mandel fractals.

FILES

`/etc/mandeld`

Executable file

SEE ALSO

Section 1, “Developing Mac OS Clients for the Network Server”

NAME

ppcd—makes Macintosh utilities available on the server and establishes connections with requesting clients

SYNOPSIS

ppcd

DESCRIPTION

ppcd runs the PPC daemon, which makes the Network Server component of Macintosh utilities available to Mac OS clients and establishes connections with requesting clients. The PPC daemon is always running on the Network Server.

The PPC daemon uses the ppcd.conf file as the basis for the information it advertises to clients. The ppcd.conf file can also be used to provide guest access.

Each Macintosh utility needs an entry in the configuration file, /etc/ppcd.conf, so that the PPC daemon can start the Network Server component of a Macintosh utility.

The file has the following format:

PPC-NAME PATHNAME SIGNATURE GUEST USERS

The fields in the ppcd.conf file must be separated by tabs.

The fields have the following values:

PPC-NAME

Contains the name of the AIX component as you want it to appear in the PPC Browser for the Macintosh client. Do not use spaces in the name.

PATHNAME

Contains the full path to the UNIX executable file.

SIGNATURE

Contains an application type or signature. The Macintosh client component uses this type to filter information in the PPC Browser. If you do not filter Network Server components from the Macintosh side in the PPC Browser, use the signature 'UNIX' as a default. The four-letter signature needs to be registered with Apple Computer, Inc.

GUEST

This field is optional. If the word guest appears in this field, any client can connect to this daemon as the guest user (an actual account). If the field is blank or contains a word other than guest, the user must supply a valid account (user name) and password in order to start the Network Server component.

USERS

This field is optional. This field contains a list of usernames for people with special access privileges.

FILES

`/etc/ppcd`

Executable file

`/etc/ppcd.conf`

Configuration file

SEE ALSO

Chapter 3, “Developing the Network Server Component”

AppleTalk services chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

ppcd.conf—the configuration file for the PPC daemon

SYNOPSIS

/etc/ppcd.conf

DESCRIPTION

The PPC daemon uses the `ppcd.conf` file as the basis for the information it advertises to clients. The `ppcd.conf` file can also be used to provide guest access.

Each Macintosh utility needs an entry in the configuration file, `/etc/ppcd.conf`, so that the PPC daemon can start the Network Server component of a Macintosh utility.

The file has the following format:

PPC-NAME PATHNAME SIGNATURE GUEST USERS

The fields in the `ppcd.conf` file must be separated by tabs.

The fields have the following values:

PPC-NAME

Contains the name of the AIX component as you want it to appear in the PPC Browser for the Macintosh client. Do not use spaces in the name.

PATHNAME

Contains the full path to the UNIX executable file.

SIGNATURE

Contains an application type or signature. The Macintosh client component uses this type to filter information in the PPC Browser. If you do not filter Network Server components from the Macintosh side in the PPC Browser, use the signature `'UNIX'` as a default. The four-letter signature needs to be registered with Apple Computer, Inc.

GUEST

This field is optional. If the word `guest` appears in this field, any client can connect to this daemon as the guest user (an actual account). If the field is blank or contains a word other than `guest`, the user must supply a valid account (user name) and password in order to start the Network Server component.

USERS

This field is optional. This field contains a list of usernames for people with special access privileges.

FILES

`/etc/ppcd`
: Executable file

`/etc/ppcd.conf`
: Configuration file

SEE ALSO

Chapter 3, “Developing the Network Server Component”

AppleTalk services chapters in *Using AIX, AppleTalk Services, and Macintosh Utilities for the Network Server*

NAME

`OF_child`—obtains the Open Firmware handle to the first child of a device in the Open Firmware device tree

SYNOPSIS

`long OF_child (`*handle*`);`

`long` *handle*`;`

DESCRIPTION

The `OF_child` routine obtains information from the Open Firmware device tree. The routine takes the Open Firmware handle (an index into the device tree) and returns the Open Firmware handle of the first child of the device.

RETURN VALUES

On successful completion, the `OF_child` routine returns the handle of the node of the child. If a child device does not exist, the routine returns 0. If an invalid *handle* value is used or the routine fails, the routine returns –1.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines

SEE ALSO

`OF_parent`(3), `OF_peer`(3), `OF_hdl2path`(3)

Chapter 6, “The Open Firmware Device Tree”

NAME

`OF_getprop`—obtains the value of an Open Firmware property from the Open Firmware device tree

SYNOPSIS

`long OF_getprop (`*handle*`,` *prop*`,` *buf*`,` *size*`);`

`long` *handle*`;`

`char *`*prop*`;`

`char *`*buf*`;`

`long` *size*`;`

DESCRIPTION

The `OF_getprop` routine obtains information from the Open Firmware device tree. The routine uses the Open Firmware handle of the device (*handle*) and the name of a property (*prop*) belonging to that handle, and copies the value of that property into the memory pointed to by *buf* up to *size* bytes. If the memory needed to store the property value is larger than the size provided, the value is truncated. To determine the memory needed to store the property value, call this routine with *size* set to 0.

RETURN VALUES

On successful completion, the `OF_getprop` routine returns the actual size, in bytes, of the property value. For string-encoded values, the null terminator is included in the returned size. If an invalid *handle* value is used or the routine fails, the routine returns –1.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines

SEE ALSO

`OF_parent`(3), `OF_peer`(3), `OF_hdl2path`(3)

Chapter 6, “The Open Firmware Device Tree”

NAME

OF_hdl2path—converts an Open Firmware handle to an Open Firmware path

SYNOPSIS

```
long OF_hdl2path (handle, buf, size)

long handle;

char *buf;

long size;
```

DESCRIPTION

The OF_hdl2path routine obtains information from the Open Firmware device tree. The routine uses the index into the device tree (*handle)* to get the corresponding fully qualified Open Firmware path of the device. This routine copies up to *size* bytes into the memory pointed to by *buf*. If the pathname is longer than the size provided, then the pathname is not null-terminated in *buf* and may be truncated. To determine the memory needed to store the path, call this routine with the *size* value set to 0.

RETURN VALUES

On successful completion, the OF_hdl2path routine returns the length, in bytes, of the full path, not including the null terminator. If an invalid *handle* value is used or the routine fails, the routine returns –1.

FILES

/usr/lib/libcfg.a

Archive of device configuration subroutines

SEE ALSO

OF_path2hdl(3)

Chapter 6, “The Open Firmware Device Tree”

NAME

`OF_nextprop`—obtains the name of the next Open Firmware property in the Open Firmware device tree

SYNOPSIS

`long OF_nextprop (`*handle*`,` *prop*`,` *buf*`)`

`long` *handle*`;`

`char *`*prop*`;`

`char *`*buf*`;`

DESCRIPTION

The `OF_nextprop` routine obtains information from the Open Firmware device tree. The routine uses an index into the device tree (*handle*) and the name of a property (*prop*) belonging to the node that the handle indicates, and returns the name of the next property in the node's property list. The function copies 0 to 31 bytes and a null character into *buf*, which results in a null-terminated name string. To obtain the name of the first property in a property list of a node, set *prop* to `NULL`. If the *prop* you specify doesn't exist or if there are no more properties to report, the routine returns a null byte in *buf*. The *buf* value should be an array of at least 32 characters.

RETURN VALUES

On successful completion, the `OF_nextprop` routine returns 1. If the routine does not find a property for the node, the routine returns 0. If an invalid *handle* value is used or the routine fails, it returns –1.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines

SEE ALSO

`OF_getprop`(3)

Chapter 6, "The Open Firmware Device Tree"

NAME

`OF_parent`—obtains the Open Firmware handle to the parent of a device in the Open Firmware device tree

SYNOPSIS

`long OF_parent (`*handle*`);`

`long` *handle*`;`

DESCRIPTION

The `OF_parent` routine obtains information from the Open Firmware device tree. The routine uses the index into the device tree (*handle)* to return the Open Firmware handle of the parent of the device.

RETURN VALUES

On successful completion, the `OF_parent` routine returns the handle of the parent in the device tree. If a parent does not exist (that is, if the node indicated by *handle* is the root node), the routine returns 0. If an invalid *handle* value is used or the routine fails, it returns –1.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines

SEE ALSO

`OF_peer`(3), `OF_child`(3), `OF_hdl2path`(3)

Chapter 6, “The Open Firmware Device Tree”

NAME

OF_path2hdl—converts an Open Firmware path to an Open Firmware handle

SYNOPSIS

long OF_path2hdl (*path*)

char **path*;

DESCRIPTION

The OF_path2hdl routine obtains information from the Open Firmware device tree. The routine uses the path into the device tree to return the device's corresponding Open Firmware handle.

RETURN VALUES

On successful completion, the OF_path2hdl routine returns the handle corresponding to the path in the device tree. If the path does not correspond to a node in the device tree, the routine returns –1.

FILES

/usr/lib/libcfg.a

Archive of device configuration subroutines

SEE ALSO

OF_hdl2path(3)

Chapter 6, "The Open Firmware Device Tree"

NAME

`OF_peer`—obtains the Open Firmware handle of the sibling of a device in the Open Firmware device tree

SYNOPSIS

`long OF_peer (`*handle*`)`

`long` *handle*`;`

DESCRIPTION

The `OF_peer` routine obtains information from the Open Firmware device tree. The routine uses the index into the device tree (*handle*) and returns the Open Firmware handle of the device's sibling, or peer.

If you specify 0 for *handle*, the routine returns the handle for the root node of the device tree.

RETURN VALUES

On successful completion, the `OF_peer` routine returns the handle of the sibling in the Open Firmware device tree. If a sibling does not exist, the routine returns 0. Otherwise, the routine returns –1.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines

SEE ALSO

`OF_child`(3), `OF_parent`(3)

Chapter 6, "The Open Firmware Device Tree"

NAME

pci_cfgrw—reads and writes PCI bus slot configuration registers

SYNOPSIS

```
#include <sys/mdio.h>

int pci_cfgrw (bid, md, write_flag)

int bid;

struct mdio *md;

int write_flag;
```

DESCRIPTION

The `pci_cfgrw` routine provides serialized access to the configuration registers for a PCI bus. To ensure data integrity in a multi-processor environment, a lock is required before accessing the configuration registers. Depending on the value of the `write_flag` parameter, a read or write to the configuration register is performed at offset `md_addr` for the device identified by `md_sla`.

The `pci_cfgrw` kernel service provides the same services for kernel extensions as the MIOPCFGET and MIOPCFPUT ioctls provide for applications. The `pci_cfgrw` kernel service can be called from either the process or the interrupt environment.

The `pci_cfgrw` kernel service is part of the Base Operating System (BOS).

OPTIONS

`bid` Specifies the bus identifier.

Set to zero on the Network Server.

`md` Specifies the address of the `mdio` structure.

The `mdio` structure contains the following fields:

`md_addr`	Specifies the full, real address of the configuration register to access (0 to 0xFF). The address is formed by adding the base address of the device and the register offset in configuration space.
`md_data`	Pointer to the data buffer.
`md_size`	Specifies the number of items of size specified by the `md_incr` parameter. The maximum size is 256.
`md_incr`	The valid access type is `MV_WORD`.

(MV_SHORT and MV_BYTE are not supported.)

md_sla	The value returned by resolve_pci_cfg_space for the device.
md_length	The bus range of the parent bus.

write_flag

Set to 1 for write and 0 for read.

RETURN VALUES

The pci_cfgrw routine returns a value of 0 when it completes successfully.

0	Indicates success
ENOMEM	Indicates that no memory could be allocated.
EINVAL	Indicates that the bus, device, function, or size is not valid.

SEE ALSO

“Machine Device Driver” in *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*

NAME

resolve_gc_offset—compute the base and dbDMA address for a child of the Grand Central (Integrated I/O controller) device

SYNOPSIS

int resolve_gc_offset (*cusobj*, *baddr*, *baddr_size*, *dbDMA_xe_addr* *dbDMA_xe_size*, *dbDMA_re_addr*, *dbDMA_re_size*)

struct CuDv * *cusobj*

ulong **baddr*;

ulong **baddr_size*;

ulong **dbDMA_xe_addr*;

ulong **dbDMA_xe_size*;

ulong **dbDMA_re_addr*;

ulong **dbDMA_re_size*;

DESCRIPTION

The resolve_gc_offset routine computes the base address and dbDMA addresses for a device that is a child of the Grand Central node. The routine resolves the addresses by adding the device offsets found in the Open Firmware device tree to the parent Grand Central node's base_addr attribute.

Using *cusobj*, this routine extracts the parent Grand Central node's base_addr atrribute. Also using *cusobj,* it extracts the device's OF_handle attribute which is then used to query the reg property in the Open Firmware device tree. For children of the Grand Central, as many as three entries can be present in the reg property. The first entry is interpreted as the device's base offset and size. The second entry, if present, is as the device's transmit dbDMA offset and size, and the third entry, if present is always interpreted as the device's receive dbDMA offset and size.

The routine returns the base address of the child device in *baddr*. The routine returns the size of the address space in *baddr_size*. If the reg property contains a transmit dbDMA offset, then the actual transmit dbDMA address of this child device is returned in *dbDMA_xe_addr* and the size of this address space is returned in *dbDMA_xe_size*. If the reg property contains a receive dbDMA offset, then the actual receive dbDMA address of this child device is returned in *dbDMA_re_addr* and the size of this address space is returned in *dbDMA_re_size*. If the reg property does not provide a dbDMA offset, then the respective address and size parameters are both set to 0. Set any of these return fields to NULL if you do not require the return data.

This routine should be used by a device method only if that device is a child of the Apple Integrated I/O (for example, the Grand Central node) in the Open Firmware device tree.

RETURN VALUES

E_OK	Indicates success
E_MALLOC	Indicates that the allocation of necessary memory storage failed.
E_NOATTR	Could not get an attibute from the database.

Other errors indicates that the address could not be resolved given the information in the OpenFirmware device tree.

E_APPLE_OFHANDLE	The node-handle appears to be invalid.
E_APPLE_NOOFREG	The `reg` property doesn't exist in the OpenFirmware device tree.
E_APPLE_OFSIZE	The `reg` property is not of the expected size.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines.

NAME

resolve_intr_lvl—reads the device and dbDMA interrupt levels for a device

SYNOPSIS

int resolve_intr_lvl (*cusobj*, *dev_intr_lvl*, *xe_intr_lvl re_intr_lvl*)

struct CuDv * *cusobj*

int * *dev_intr_lvl*;

int * *xe_intr_lvl*;

int * *re_intr_lvl*;

DESCRIPTION

The resolve_intr_lvl routine reads the integer values from the Open Firmware device tree that identify a device's interrupt source. This routine uses *cusobj* value to extract the device's OF_handle attribute and use it to query the "AAPL,interrupts" property in the Open Firmware device tree. If the device does not have an "AAPL,interrupts" property, then the routine traverses the device tree looking for a parent node from which the device can inherit its interrupt source. (This situation is likely to happen for PCI bridge devices.) Devices may have several possible interrupt sources. The first interrupt value is always that of the device and is returned in *dev_intr_lvl*. The existance of any following values represent the DMA interrupts. The DMA transmit interrupt level, if present, is returned in *xe_intr_lvl* and the DMA receive interrupt level, if present, is returned in *re_intr_lvl*.

If the "AAPL,interrupts" property does not provide a DMA interrupt level, then *xe_intr_lvl* and *re_intr_lvl* are both set to -1. Set any of these return fields to NULL if you do not require the return data.

This routine can be used by any device method that expects its device to have an "AAPL,interrupts" propety in the Open Firmware device tree, or wants to inherit the "AAPL,interrupts" property from a parent node.

RETURN VALUES

E_OK	Indicates success
E_NOATTR	Could not get an attribute from the database.

Other errors indicates that the interrupts could not be interpreted given the information in the OpenFirmware device tree.

E_APPLE_OFHANDLE	The node-handle appears to be invalid.
E_APPLE_NOOFINTR	The "AAPL,interrupts" property doesn't exist in the OpenFirmware device tree.
E_APPLE_OFSIZE	The "AAPL,interrupts" property is not of the expected size.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines.

NAME

`resolve_pci_cfg_space`—obtains the configuration space address for a child of the PCI device

SYNOPSIS

`int resolve_pci_cfg_space (`*handle*`,` *cfg_addr*`)`

`long` *handle*`;`

`ulong *`*cfg_addr*`;`

DESCRIPTION

The `resolve_pci_cfg_space` routine obtains information from the Open Firmware device tree. The routine uses a device's Open Firmware *handle* to find the configuration space address that has been assigned to that device in *cfg_addr*. The routine searches the first entry of the device's `reg` property in the Open Firmware device tree for the configuration space.

This routine should be used by a device method only if that device is a child of a PCI bus or PCI bridge in the Open Firmware device tree.

RETURN VALUES

`E_OK`	Indicates success
`E_MALLOC`	Indicates that the allocation of necessary memory storage failed.

Other errors indicates that the address could not be resolved given the information in the OpenFirmware device tree.

`E_APPLE_OFHANDLE`	The node-handle appears to be invalid.
`E_APPLE_NOOFREG`	The `reg` property doesn't exist in the OpenFirmware device tree.
`E_APPLE_OFSIZE`	The `reg` property or the assigned-addresses property is not of the expected size.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines.

NAME

`resolve_pci_io_space`—obtains the physical, 32-bit I/O space address for a child of a PCI device

SYNOPSIS

`int resolve_pci_io_space (`*handle*`,` *io_addr*`,` *io_addr_size*`)`

`long` *handle*`;`

`ulong *`*io_addr*`;`

`ulong *`*io_addr_size*`;`

DESCRIPTION

The `resolve_pci_io_space` routine obtains information from an Open Firmware device tree. The routine uses a device's Open Firmware *handle* to access the device information in the device tree. The function returns the absolute, 32-bit I/O space address in *io_addr* (within the PCI domain's address space) that has been assigned to that device. It also returns the size in bytes of that assigned space in *io_addr_size*. The routine uses the `OF_getprop`() function to read up the "`reg`" and "`assigned-addresses`" properties from the Open Firmware device tree. It searches for the first entry of the "`reg`" property that indicates a 32-bit I/O space address. If that entry identifies a relocatable region, then the region's corresponding assigned physical address and size is searched for in the "`assigned-addresses`" property.

This routine should be used by a device method only if that device is a child of a PCI bus or PCI bridge in the Open Firmware device tree. This routine deals only with primary access paths within the "`reg`" property; secondary access paths listed in the "`alternate-reg`" property are not used.

RETURN VALUES

`E_OK`	Indicates success
`E_MALLOC`	Indicates that the allocation of necessary memory storage failed.

Other errors indicates that the address could not be resolved given the information in the OpenFirmware device tree.

`E_APPLE_OFHANDLE`	The node-handle appears to be invalid.
`E_APPLE_NOOFREG`	The reg property doesn't exist in the OpenFirmware device tree.

`E_APPLE_OFSIZE`	The reg property or the assigned-addresses property is not of the expected size.
`E_APPLE_PHYSADDR`	A relocatable address found in the reg property does not have a corresponding assigned-address entry giving the true physical address.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines.

NAME

`resolve_pci_mem_space`—obtains the physical, 32-bit memory space address for a child of a PCI device

SYNOPSIS

`int resolve_pci_mem_space (`*handle*`,` *mem_addr*`,` *mem_addr_size*`)`

`long` *handle*`;`

`ulong *`*mem_addr*`;`

`ulong *`*mem_addr_size*`;`

DESCRIPTION

The `resolve_pci_mem_space` routine obtains information from the Open Firmware device tree. This routine uses a device's Open Firmware *handle* to returns the absolute, 32-bit memory space address (within the PCI domain's address space) that has been assigned to that device in *mem_addr*. The routine also returns the size in bytes of that assigned space in *mem_addr_size*. The routine uses the `OF_getprop`() routine to read the "`reg`" and "`assigned-addresses`" properties from the device tree. The `resolve_pci_mem_space` routine searches the "`reg`" property for the first entry indicating a 32-bit Memory Space address. If that entry identifies a relocatable region, then the region's corresponding assigned physical address and size will be searched for in the “`assigned-addresses`” property.

This routine should be used by a device method only if that device is a child of a PCI bus or PCI bridge in the Open Firmware device tree. This function deals only with primary access paths within the "`reg`" property; secondary access paths listed in the "`alternate-reg`" property are not used.

RETURN VALUES

`E_OK`	Indicates success
`E_MALLOC`	Indicates that the allocation of necessary memory storage failed.

Other errors indicates that the address could not be resolved given the information in the Open Firmware device tree.

`E_APPLE_OFHANDLE`	The node-handle appears to be invalid.
`E_APPLE_NOOFREG`	The `reg` property doesn't exist in the Open Firmware device tree.
`E_APPLE_OFSIZE`	The `reg` property or the assigned-addresses property is not of the expected size.

`E_APPLE_PHYSADDR`	A relocatable address found in the `reg` property does not have a corresponding assigned-address entry giving the true physical address.

FILES

`/usr/lib/libcfg.a`

Archive of device configuration subroutines.

Appendix   Keyboard Positions

This appendix describes differences between Apple and IBM keyboards and describes the differences in key positions for foreign keyboards.

For information about AIX keyboards and detailed diagrams of the keyboards, see the AIX *Technical Reference*. You can use this reference through InfoExplorer.

The Network Server Keyboard

The Network Server uses the 101-key keyboard, ISO 8859-1. The following figure shows the keyboard and the numbered position codes for each key.

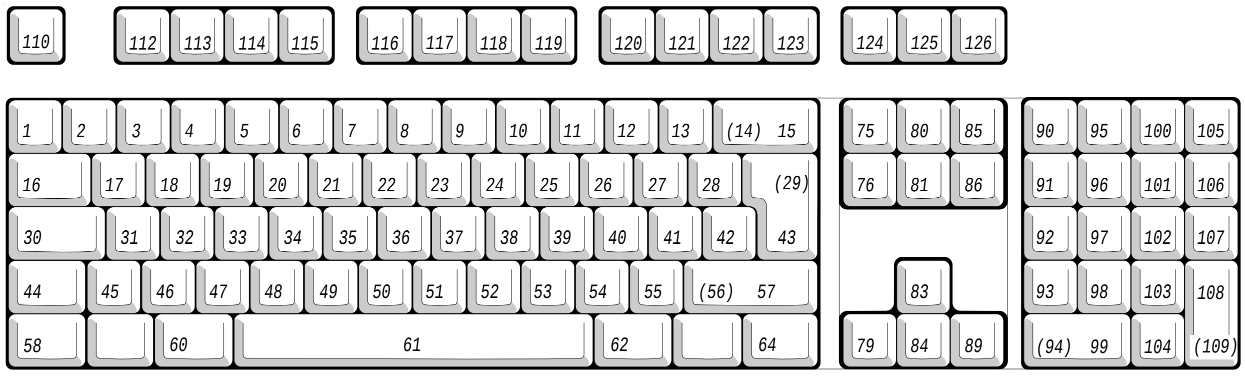


Figure A-1 The key positions for the Network Server

The following sections detail differences in key positions for international keyboards.

Generating the third symbol for a key

Most Apple keyboards only have two symbols printed on each key whereas IBM keyboards provide three symbols per key. To generate the third symbol on a key, hold down the Alternate Graphic (the Apple Command key, position number 60 or 62) and press the key you want.

Creating diacritical marks

Diacritical marks (or “dead keys”) are keys that do not print out when you type them; they print when you type the next character. Diacritical marks include accents, umlauts, and circumflexes. For example, to add diacritical mark to a letter, type the dead key and then type the letter for the accent. When you type the dead key, nothing appears. When you type the next key, it appears with the accent above it. The tables in the following sections list the key positions for many dead keys.

Differences for international keyboards

This section highlights the differences in key positions for international keyboards. The international keyboards use the same 101-key keyboard (ISO 8859-1), however, for many countries the some keys are in different places.

For all international Apple keyboards, keys on the numeric keyboard have the following mappings:

Key Position	Character
95	=
100	/
105	*
106	-
107	+

For more information about keyboards and keyboard layouts, see the *AIX Technical Reference* in InfoExplorer.

Italian Keyboard

The following table details the differences for the key positions between the IBM Italian keyboard and the Apple Italian keyboard. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	@	#	
2	&	1	
3	"	2	
4	'	3	
5	(	4	
6	ç	5	
7	è	6	
8	)	7	{
9	£	8	[
10	à	9	]
11	é	0	}
12	-	_	
13	=	+	
17	q	Q	\
18	z	Z	
27	ì	ˆ	
28	$	*	~
40	m	M	
41	ù	%	
42	§	°	‘
46	w	W	
52	,	?	
53	;	.	
54	:	/	
55	ò	!	

United Kingdom English Keyboard

The following table details the differences for the key positions between the IBM United Kingdom English keyboard layout and the Apple United Kingdom keyboard. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	§	±	
3	2	@	2
7	6	ˆ	
8	7	&	
9	8	*	
10	9	(	
11	0	)	°
12	-	_	#
13	=	+	¬
27	[	{	
28	]	}	‾ (overbar)
40	;	:	
41	'	"	
42	\	|	
45	‘	~	

Finnish/Swedish Keyboard

The following table details the differences for the key positions between the IBM Finnish/Swedish keyboard and the Apple Finnish/Swedish keyboard. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	§	°	1/2

Norwegian Keyboard

The following table details the differences for the key positions between the IBM Norwegian keyboard and the Apple keyboard. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	'	§	
3	2	"	|
5	4	$	Int. Currency Symbol
13	’	‘	\
42	@	*	

Danish Keyboard

The following table details the differences for the key positions between the IBM Danish keyboard and the Apple Danish keyboard. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	$	§	
5	4	Int. Currency Symbol	1/2

Belgian-French, Dutch, and Flemish Keyboards

The following table details the differences for the key positions between the IBM Belgian-French, Dutch, and Flemish keyboard and the Apple Belgian-French, Dutch, and Flemish keyboards. See the keyboard layout map at the beginning of this appendix to locate the key positions.

Key Position	Base Character	Shift Character	Alternate Graphic
1	@	#	
3	é	2	^{2}
4	"	3	^{3}
42	‘	£	µ

Index