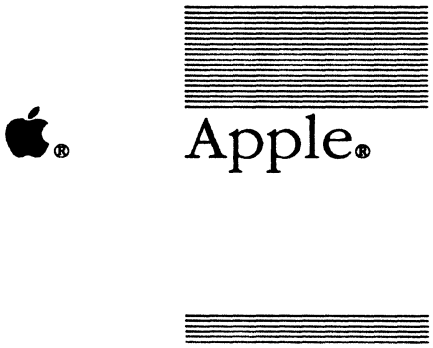


A/UX® Network Applications Programming

Beta/Final Draft

09/19/88

Michael Hinkson

Developer Technical Publications

This document contains preliminary information.

It does not include:

- final technical information
- final editorial corrections
- an index
- Final art

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

WARNING

This equipment has been certified to comply with the limits for a Class B computing device pursuant to Subpart J of Part 15 of FCC rules. Only peripheral devices (computer input/output devices, terminals, printers, and so on) certified to comply with Class B limits may be attached to this computer.

Operation with noncertified peripheral devices is likely to result in interference to radio and television reception.

Chapter 1
Introduction

Contents

Figures

Chapter 1

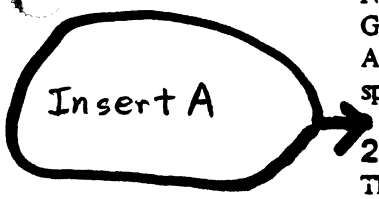
Introduction

1. About this manual

This manual documents the A/UX networking capabilities at the level required to write network software applications.

For the B-NET Transport Control Protocol/Internet Protocol (TCP/IP) networking implementation, software applications can access the network through the socket abstraction at the level of interprocess communication (IPC). At this level A/UX is basically compatible with 4.3 BSD networking. Chapters 2 and 3 of this manual are the introductory and advanced 4.3 BSD tutorials on IPC, modified to document only what A/UX supports.

For the Network File System (NFS) implementation, software applications can access remote systems by using the remote procedure call (RPC). At this level A/UX uses Sun Microsystems' Release 3.0 NFS. Chapter 4 of this manual is Sun's Release 3.0 RPC Programming Guide, modified to reflect A/UX-specific information where required. Appendixes A through D contain the Release 3.0 protocol specifications.



2. Terminology

The following terms are used in this manual.

host A machine on the network; often referred to as *local host* and *remote host*.

node A machine on the network. Network design may be described using graph theory, in which the component machines (microprocessors, printers, terminals) are viewed as nodes, connected by arcs.

socket A logical abstraction. A socket may be implemented in many different ways, but is generally used to establish a

Insert A, page 1-1

For the AppleTalk® implementation, software applications can access the printing services of AppleTalk-capable printers (LaserWriter® printers, for example) over Apple's low-cost LocalTalk™ cable system. Chapter 5 of this manual presents a programmer's overview of the A/UX implementation of AppleTalk with a description of the supported protocols and several programming examples.

connection.

protocol A well-known set of conventions (about how data is represented, checked, transmitted, and so forth) that must be implemented at both ends of a connection before any communication can take place.

protocol layers
A modular implementation of protocols, in which each "layer" hides the details of its functioning from the user as well as from other layers. Each protocol layer is built on top of its predecessor. Protocol layers allow for peer-to-peer communication, with processes from the same protocol layer on two different machines talking to each other. See "The OSI Model" later in this chapter for more information.

internet A group of networks interconnected by gateways or bridges (or both). The word **Internet** when capitalized and used as a noun, usually refers to the Defense Data Network (DDN), heir to the networking reasearch and development performed on the DARPA internet (also called ARPANET). When capitalized and used as an adjective (for example, Internet domain), Internet refers to a standard used by the DDN.

gateway A connector between two or more different types of network.

bridge A connector between two similar networks.

3. Overview of networking concepts

This section defines networking concepts used in this manual.

3.1 Connectivity

Connectivity denotes a type of network service ranging from connected to connectionless. A protocol is said to provide **connected service** when the protocol layer "knows" about an exchange of information between two parties. A standard analogy for connected service is a telephone call; a wire completes the circuit, and the telephone company "knows" about the exchange between two parties.

A protocol provides **connectionless service** when there is no established connection, as for example, when packets are broadcast on the network. Each communication delivered is considered a separate job accomplished.

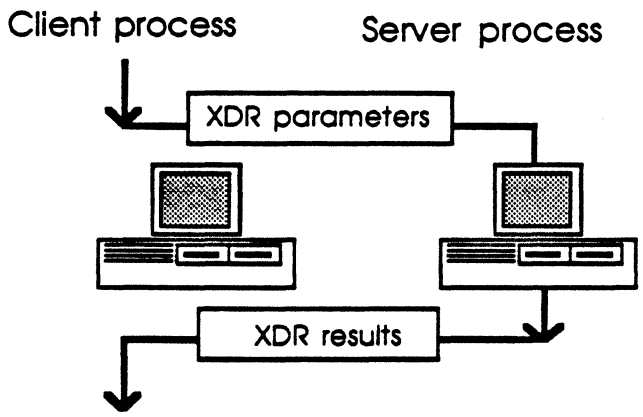
3.2 The client/server model

The **client/server model** is a commonly used paradigm in constructing distributed applications: **client** applications request services from a **server** process. That is, the client actively initiates communication while the server passively "listens" on its socket.

Although this paradigm has been extended to hardware servers, the model referred to here is that of client process and server process. Depending on whether a protocol is symmetric or asymmetric, client and server processes may be able to switch roles. In a **symmetric protocol**, either side may play the server or client role. An **asymmetric protocol** is one in which the client always initiates communication.

For example, in the NFS software, the client invokes a remote procedure call. The remote procedure call library encodes data in external data representation (XDR) form, as illustrated in Figure 1-1. The server, in turn, responds to the request by sending back an XDR result.

Figure 1-1. The client/server model



Insert B

3.3 Addressing versus naming

Network hosts or entities may be accessed by using network addresses or a host name or entity name. **Addressing** uses the network address; for the B-NET software, this is a four-byte internet address expressed in decimal or hexadecimal form.

When **naming** is used, a host name is translated into the network number with tables (such as the `/etc/hosts` file for the B-NET software), served databases (such as the yellow pages distributed database), or the Internet name domain service. Higher level protocols allow you to use names as well as numbers.

3.4 Data integrity

Data integrity refers to how much and what type of checking is done to establish whether data has been received, if it has been corrupted, or if parts of the data are out of order. Generally speaking, there are two types of sockets: low data integrity and high data integrity sockets. For example, a **stream socket** is a high data integrity socket that provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. A **datagram socket** is a low data integrity socket that supports bidirectional flow of data that is not guaranteed to be sequenced, reliable, or unduplicated.

3.5 Error detection

Error detection is the task of establishing that data is corrupted or out of sequence. When an error is detected, the user is informed of it.

3.6 Error recovery

Error recovery is the task of recompiling the data into its proper order and/or integrity. As this is a much bigger job than merely detecting the occurrence of errors, protocols may provide error detection without attempting to supply the user with error recovery functions.

3.7 Flow control

Flow control is the regulation of the passage of data, so that only a certain portion is sent at a time. This is useful to ensure that the network is not overloaded with data.

Insert B, page 1-4

Network hosts or entities may be accessed by using network addresses or a host name or entity name. **Addressing** uses the network address. For TCP/IP, this network address is a four-byte internet address expressed in decimal or hexadecimal form. For AppleTalk, this network address is a 32-bit number comprising an internet address, a network address, and a node address.

When **naming** is used, a host name or entity name is translated into the network number in one of several ways:

- with tables, such as the `/etc/hosts` file for TCP/IP, or the Names Table for AppleTalk
- with served databases (such as the Yellow Pages distributed database or the AppleTalk Names Directory
- by the Internet name domain service.

Higher-level protocols allow you to use names as well as numbers.

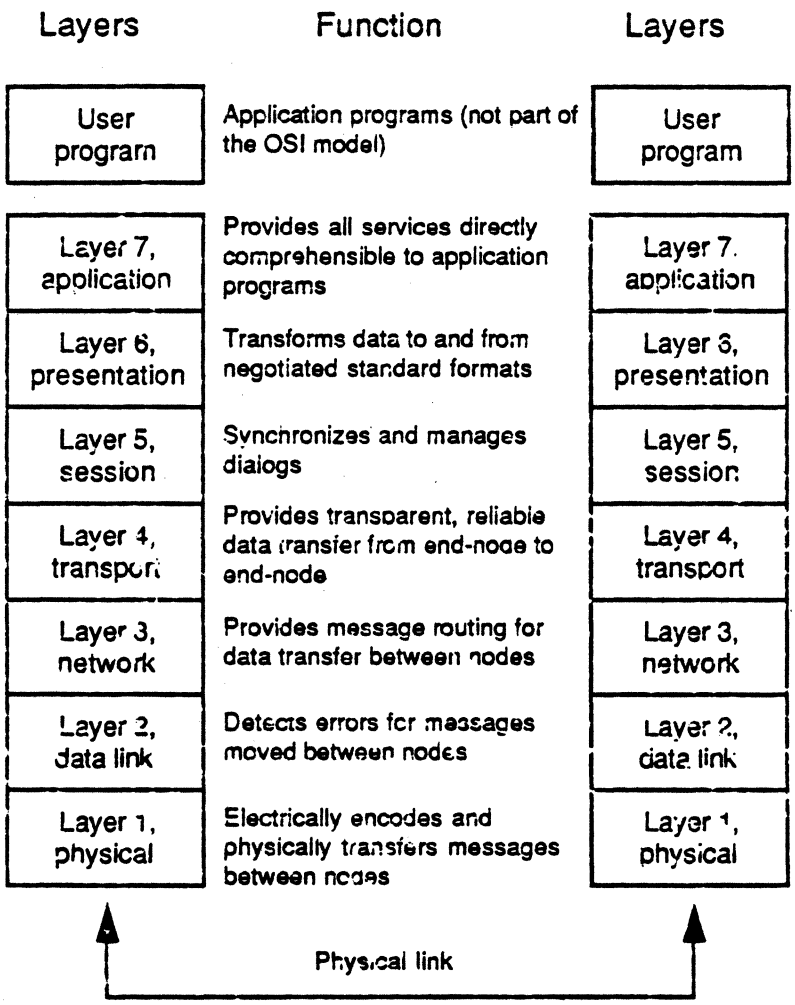
4. The OSI model

The International Standards Organization (ISO) seven-layer Open Systems Interconnection (OSI) model is a model for network protocol layers and their interworkings.

The OSI model defines an open architecture that uses accepted standard protocol layers. It presumes a modularization of network support software based on a layer function. Each module forms a layer in the model and is responsible for providing selected network services to the layer above. These services are provided by functions performed within that layer and through services available from the layer(s) below.

Each layer in the OSI model describes a type of required functionality and is assumed to communicate with the same layer on another machine, as illustrated in Figure 1-2. Thus, the software for any layer may be replaced with a new version without affecting the user's perception of network operation.

Figure 1-2. The OSI reference model



The OSI model is concerned not only with reliable data transfer, but also with providing network functions to the user. These are supplied by layer 7, the application layer, which provides the user with functions such as file transfer, electronic mail, virtual terminals, and procedures that can be called by user programs.

To guarantee the internetworking of cooperating end systems (that is, devices that contain all seven layers of the OSI model), a common language or data representation must be used so that the cooperating end systems can understand each other. This common data representation is provided by layer 6, the presentation layer.

Another factor needed to guarantee the internetworking of end systems is control over the way a conversation between two systems is conducted. This conversation control is provided by layer 5, the session layer.

The actual movement of data between end systems is provided by layers 4 through 1 of the OSI model. Layer 4, the transport layer, provides for reliable end-to-end transfer of messages between end systems. Layer 3, the network layer, provides for the routing and relaying of messages between end systems of the network layer. Layer 2, the data link layer, provides for error detection and correction of packets moved between end systems. Layer 1, the physical layer, arbitrates access to the physical network, electrically encodes packets, and physically transmits the packets across the physical network. The actual cable, although not a part of the OSI reference model, is often included in layer 1.

Each layer of the OSI model on an end system is the peer of its corresponding layer on another end system. The advantage of peer-to-peer over master-slave communications is the ability of communicating end systems to negotiate protocol options with each other. There is a one-to-one relationship between layers, whether the transfer of data is directly between end systems or between end systems that must traverse intermediate systems to reach each other.

Each layer requests services from the layer below. Two types of information, control information and data, are passed between layers in providing these services. The control information is the basis for all

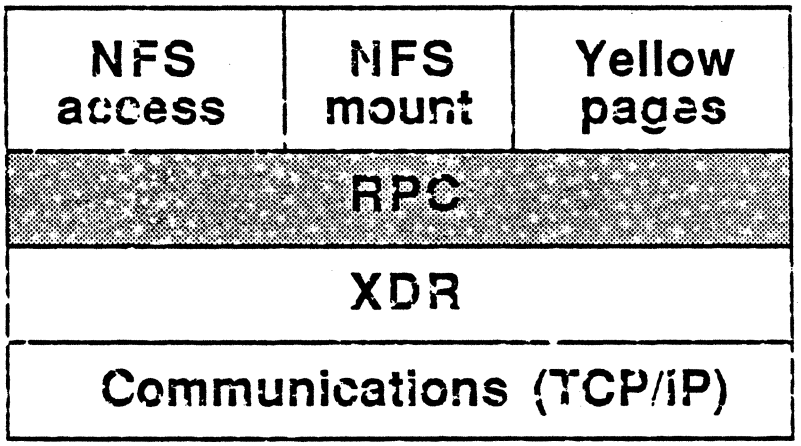
the services that are required to process the message. As each layer provides its part of those services, the remaining control information is passed to the next lower layer.

The data passed down to a lower layer is generally transported unchanged. An exception to this happens in the presentation layer, where data is reformatted. Each layer prefaces the data with control information before requesting the services of the next lower layer. This control information is interpreted by the corresponding layer in the receiving end system.

4.1 NFS and yellow pages facilities

The relation of the NFS and yellow pages facilities to each other and to the network is shown in Figure 1-3.

Figure 1-3. NFS and yellow pages facilities on the network



The XDR, RPC, and NFS protocols are all built on top of each other and on top of B-NET's TCP/IP communication facilities. Each of the NFS services accesses the underlying structure.

Insert C →

Insert C, page 1-8

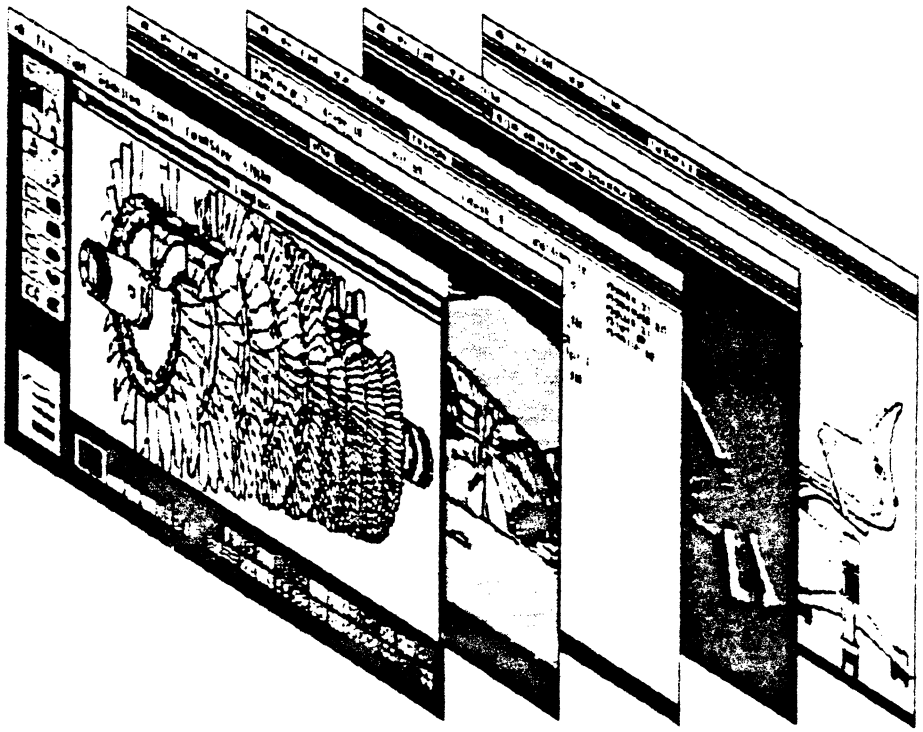
4.2 AppleTalk

The AppleTalk protocols correspond roughly to the OSI model. Chapter 5 of this manual explains this correspondence in detail.

Part 2

Changes to A/UX Man Pages for AppleTalk

Apple® A/UX™ System Administrator's Reference



NAME

appletalk — configure and view AppleTalk® network interfaces

SYNOPSIS

appletalk [-i *interface* [-n] [-u] [-d] [-s] [-t] [-r [*RTS-attempts*]]

DESCRIPTION

appletalk is a utility for configuring and/or viewing AppleTalk network interfaces and the AppleTalk network. appletalk can be used at any time to view network interface parameters, or to bring up or down an AppleTalk interface.

The following arguments are supported.

-i *interface* — The *interface* parameter defines the interface to configure or view; this parameter is a string such as localtalk0. appletalk defaults to the DDP (Datagram Delivery Protocol) interface defined in appletalkrc(4).

-n — Displays the AppleTalk interface initial and current node addresses. If the AppleTalk network has not been activated, these addresses will be zero. The initial address is the address which this interface first uses during the ALAP (AppleTalk Link Access Protocol) dynamic node assignment phase. Once a unique address is found, it is saved for use as this interface's initial node address. The current address is the unique AppleTalk logical address assigned to this interface.

-u — Brings this interface online (up). If this interface is the DDP interface (as specified in appletalkrc(4)), the AppleTalk NBP (Name Binding Protocol) daemon (/etc/at_nbpd) will be started as well. You must be the superuser to use this option.

-d — Brings this interface offline (down). If this interface is the DDP interface (as specified in appletalkrc(4), the AppleTalk NBP daemon (/etc/at_nbpd) will be shut down as well. You must be the superuser to use this flag option.

`-s` Displays statistics and error count for this interface. If the AppleTalk network is active, DDP statistics and error count will also be displayed.

`-t` Displays the ALAP types registered on this interface. By default, DDP short and long datagram types are registered on circuits 1 and 2.

`-r` *RTS-attempts* (LocalTalk™ interfaces only). Displays the number of RTS attempts to make when transmitting an ALAP frame. Supplying an optional numeric parameter sets the number of RTS attempts to *RTS-attempts*. You must be the superuser to use this *RTS-attempts* option.

EXAMPLES

```
appletalk -i locatalk0 -u
appletalk -s
```

The first command brings the interface `localtalk0` online; the second displays statistics and error counts for the DDP interface.

FILES

```
/etc/appletalk
/etc/appletalkrc
/dev/appletalk/ddp/socket
/dev/appletalk/lap/*/control
/etc/at_nbpd
```

SEE ALSO

`appletalkrc`(4), `appletalk`(7); "Installing and Administering AppleTalk," in *A/UX Network System Administration.*

NAME

`fwdload` — load an application onto an intelligent peripheral

SYNOPSIS

`fwdload [-a] [-v] [-f`*dev*`] [-n`*name*`]` *file*

DESCRIPTION

The utility `fwdload` loads a program onto an intelligent peripheral. The peripheral must have a "forwarder" configured for it (see `forwarder`(7). If the `-f` flag option is used, the peripheral is *dev;* otherwise, standard output will be used.

The `-n` flag option allows custom names, otherwise the *file* name will be used.

The `-v` flag option provides diagnostics in verbose format.

The parameter *file* is the application to download and is in COFF format. Before the download, a reset is issued.

If the `[-a]` is used, there is no reset. Once the load is complete, execution of the downloaded application will begin at the START indicated by the COFF file.

EXAMPLE

```
fwdload -f /dev/icp13 at_load
```

will download the AppleTalk® driver onto the default AppleTalk peripheral.

FILES

```
/etc/fwdload
/etc/startup.d/fwdicp.d/at_load
/etc/startup.d/fwdicp.d/tt_load
```

SEE ALSO

`fwd_lkup`(1M), `forwarder`(7); "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

fwd_lkup — look up the application that is loaded onto a Front End Processor

SYNOPSIS

fwd_lkup [-f*dev*] [-v]

DESCRIPTION

fwd_lkup looks up the name of the application loaded onto a Front End Processor. The FEP must have a "forwarder" configured for it. If the -f flag option is used, the peripheral is *dev;* otherwise standard input will be used.

The -v flag option provides diagnostics in verbose format.

EXAMPLE

```
fwd_lkup -f /dev/fwdicp13
```

will find out what application is running on the ICP card in slot 13. If it is currently running AppleTalk®, it will print the following:

```
begin start name
0     0     at_load
7fff  0     AVAIL
7fff  0     END
```

This indicates that at_load, the AppleTalk load module is loaded on the ICP, and that it is occupying all 7fff bytes of the ICP's memory.

FILES

/usr/bin/fwd_lkup

SEE ALSO

fwdload(1M), forwarder(7); "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

newunix — prepare for new kernel configuration

SYNOPSIS

/etc/newunix [bnet] [nfs] [nonet] [toolbox] [notoolbox] [localtalk] [notalk] [tc] [notc] [slip] [noslip] [asttty] [noasttty]

DESCRIPTION

newunix begins the process of configuring a new kernel by installing (or uninstalling) the appropriate scripts and driver object files needed by autoconfig(1M). The appropriate argument to newunix depends on the type of kernel desired:

bnet
basic networking

nfs
Network File System

nonet
non-networking

toolbox
A/UX toolbox

notoolbox
no toolbox capabilities

The arguments you specify also depend on the optional peripherals and interfaces you desire:

localtalk
LocalTalk™ support

notalk
no LocalTalk™ support

asttty
support for serial port expansion board

noasttty
no support for serial port expansion board

tc support for the tape catridge

notc
no support for the tape catridge

slip
support for the Serial Line Internet Protocol

`slip`
no support for the Serial Line Internet Protocol

In order to complete the kernel configuration process, `autoconfig`(1M) should be run after `newunix`.

EXAMPLES

To prepare an NFS kernel, use

```
/etc/newunix nfs
```

To add the software that supports the tape controller to the kernel, use

```
/etc/newunix tc
```

After adding the tape controller software, should you decide to remove the tape controller software from the kernel, use

```
/etc/newunix notc
```

In all three examples above, after running `newunix` run `autoconfig` to create a new kernel and then reboot to begin using the new kernel.

FILES

`/etc/boot.d/*`	driver object files
`/etc/install.d/*`	installation scripts
`/etc/master.d/*`	script files
`/etc/startup.d/*`	startup programs
`/etc/uninstall.d/*`	uninstallation scripts
`/etc/init.d/*`	initialization scripts

SEE ALSO

`autoconfig`(1M), `finstall`(1M).
"Installing and Administering AppleTalk," in *A/UX Network System Administration.*

NAME

appletalk — general AppleTalk socket interface and I/O and STREAMS modules controls

DESCRIPTION

This manual page describes the AppleTalk I/O control calls (see ioctl(2)), device files, and the general nature of the A/UX AppleTalk interface.

Before beginning, several points should be noted. The AppleTalk library routines automatically set up and invoke the correct ioctl requests that are necessary for most AppleTalk requirements. While the ioctls give the programmer more control than the AppleTalk library routines, they require a much greater understanding of the A/UX implementation of AppleTalk. In addition, AppleTalk ioctl calls are subject to change, while AppleTalk library functions will not change. It is, therefore, strongly recommended that the library routines be used whenever possible instead of the more complicated ioctl calls.

AppleTalk itself is implemented as a series of protocol layers built into STREAMS drivers and modules. Each layer is built on top of (and uses) the previous layer. The order of layers, from lowest (closest to the physical transport) to highest (closest to the application), is AppleTalk Link Access Protocol (ALAP); Datagram Delivery Protocol (DDP); AppleTalk Transaction Protocol (ATP); and Printer Access Protocol (PAP), Name Binding Protocol (NBP), and Zone Information Protocol (ZIP) (in the same layer).

The lower layers (ALAP/DDP) are normally used only for new network testing and development, such as building a new layer using TCP/IP on top of DDP. To reduce system call overhead, the final—new—layer is best completed as an additional STREAMS module or driver to be configured into the existing kernel/FEP code.

Note: Module/driver work is not recommended except for the most experienced A/UX programmers, and the information necessary to accomplish this task is beyond the scope of this manual page.

Required Reading

See *Inside AppleTalk* (published by Apple Computers), "AppleTalk Programming Guide," in *A/UX Network Applications Programming* and at_ident(3N), atp(3N), ddp(3N), lap(3N), nbp(3N), pap(3N), and zip(3N) in *A/UX*

Programmer's Reference for details for definitions and use of the specific AppleTalk protocols and AppleTalk library routines.

The A/UX AppleTalk interface uses STREAMS and you should be familiar with AT&T's *UNIX System V STREAMS Programmer's Guide*. It also uses special Front End Processor (FEP) communications software of which the programmer should have some working knowledge; see `forwarder`(7) in *A/UX System Administrator's Reference* for more information.

STREAMS ioctl Calls

Because AppleTalk under A/UX is implemented with AT&T-style STREAMS, AppleTalk modules must be controlled with STREAMS-style ioctl calls.

To review, a standard `ioctl`(2) call is made as follows:

```
int ioctl(fd, request, arg)
int fd, request;
char *arg;
```

To turn this standard ioctl call into a STREAMS ioctl call, the AppleTalk socket value is supplied as *fd* (see "AppleTalk Sockets," below). The *request* argument is set with the token `I_STR` (defined in `/usr/include/sys/stropts.h`), and *arg* is a pointer to a `strioctl` structure.

`strioctl` is defined as follows in `/usr/include/sys/stropts.h`

```
struct strioctl {
    int ic_cmd;         /* streams ioctl request */
    int ic_timout;      /* ACK/NAK timeout */
    int ic_len;         /* length of data arg */
    int ic_dp;          /* pointer to data arg */
}
```

The user must prepare the data, allocate and fill the `strioctl` structure, and then make the standard ioctl call. See *UNIX System V STREAMS Programmers Guide* for more information on using STREAMS ioctl calls.

Note: Remember that all the ioctl tokens described below (unless otherwise noted) are STREAMS ioctls, and are passed in the `ic_cmd` field of the `strioctl` structure.

Also note that when the standard ioctl call hits the stream head it becomes a streams ioctl call.

AppleTalk Sockets

A process uses a socket as the end point of communication in sending and receiving data. In AppleTalk under the Macintosh operating system, a socket is a DDP endpoint. Under A/UX, however, a socket becomes a file descriptor that gives you access to AppleTalk resources. These file descriptors are called "AppleTalk sockets."

> *Note:* Note that the AppleTalk socket ID is different from the file descriptor that it returns. For example, if you open the special file `/dev/appletalk/ddp/socket21`, the file descriptor returned is most likely *not* 21.

A process must "own" an AppleTalk socket to use it; it acquires ownership of the socket by requesting it via one of several possible AppleTalk library `open` calls (see "AppleTalk Programming Guide"). A process can request a "static" AppleTalk socket assignment by giving the AppleTalk socket's complete path (if using the standard A/UX `open`; see `open`(2) in *A/UX Programmer's Reference*), or by supplying the AppleTalk socket number to one of the AppleTalk library `open` calls.

A process can also request a "dynamic" AppleTalk socket assignment; in this case, a free AppleTalk socket number (a file descriptor) is returned to the user. This is done by doing an `open`(2) on the file `/dev/appletalk/ddp/socket`, or by passing a socket value of zero to one of several AppleTalk library `open` calls. By definition, each AppleTalk socket aquired this way returns a single unique file descriptor.

> *Note:* The ALAP layer does not use AppleTalk socket although it is accessed through them. ALAP delivers data node-to-node only via LocalTalk. Its AppleTalk library `open` call does, however, return a standard file descriptor that is used to access the ALAP streams driver. Only the DDP and higher level layers use AppleTalk sockets.

AppleTalk socket are identified by a number in the range of 1 through 254. Numbers in the range of 1 through 63 (static sockets) are reserved; socket numbers 64 through 127 ("experimental" static sockets) are reserved for system developers, but are not to be used in final applications. Numbers ranging from 128 to 254 (dynamic sockets) are generally available for use. The values 0 and 255 are reserved and used by AppleTalk for special cases.

See *Inside AppleTalk* for more information. The C header files in `/usr/include/at` contain defines for the values of these sockets (see <at/ddp.h>).

An AppleTalk socket's internet address is comprised of a network number, a node number, and an AppleTalk socket number; it is formed by enclosing the values in braces. For example,

```
struct nbp_addr a;
        .
        .
        .
a = {10,40,128}
```

Because the numeric addresses are awkward to use and can vary from time-to-time, AppleTalk provides a mechanism for specifying objects by name instead of by number. See the sections on the Name Binding Protocol in "AppleTalk Programming Guide" and *Inside AppleTalk* for more information.

A final note on AppleTalk sockets, most of the functions that an AppleTalk applications programmer is likely to need are implemented via AppleTalk library routines and STREAMS ioctl calls. The few "normal" I/O functions are done via the standard A/UX `read`(2) and `write`(2) system calls.

The AppleTalk Model

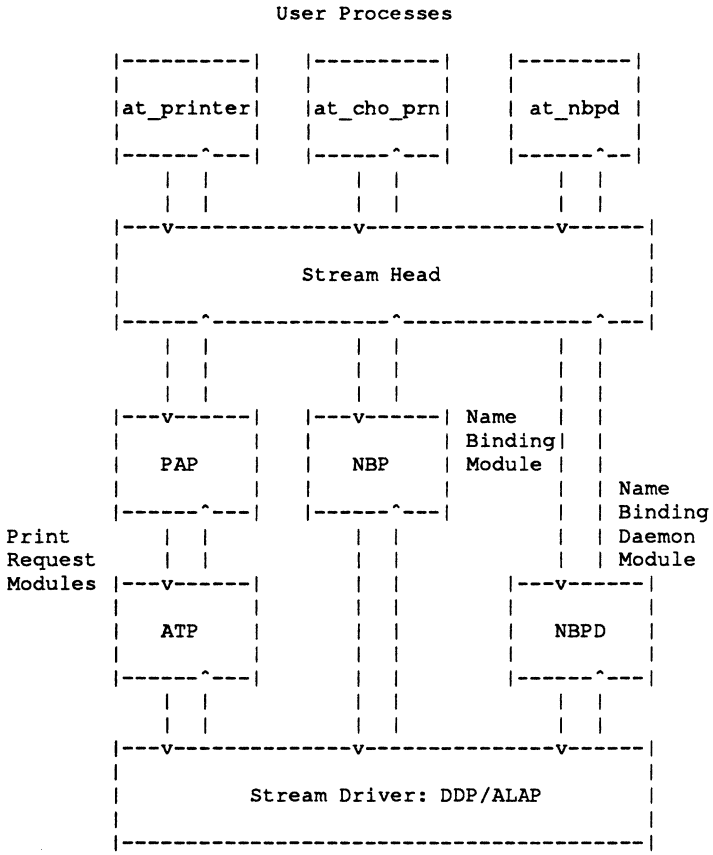
The following diagram describes the A/UX implementation of AppleTalk and gives the programmer some idea of how to program in its environment.

As previously stated, most AppleTalk protocol layers are implemented as STREAMS modules. The two exceptions are the DDP and ALAP layers. The majority of applications require the programmer to push one or more modules into the open stream in order to achieve the proper layering for that application.

The first application illustrated (`at_printer`) shows the configuration for communicating with a network print server. Note that the ATP module must be pushed before the PAP module. While it is possible to reverse the pushing order, unpredictable results can occur if this is done.

The second and third applications (`at_cho_prn` and `at_nbpd`) are normally used together. When AppleTalk is brought up a special application daemon, `nbpd`, is invoked. It opens an AppleTalk socket and pushes the module `at_nbpd` into

the stream. This "application" is used by subsequent applications, such as `at_nvelkup`(1), to open a socket and push the module `at_nbp` into the stream. Modules `at_nbp` and `at_nbpd` communicate at the ALAP level to complete users requests for name binding information.



THE APPLETALK PROTOCOLS

The AppleTalk protocols use STREAMS ioctls (`I_STR`) which take buffer (`ic_dp`), length (`ic_len`), and timeout (`ic_timout`) values as parameters. As noted in "STREAMS ioctl Calls," these fields are defined as part of the `ioc_cmd` structure.

Note: Remember that library routines are provided which

feed the correct parameters to the ioctls for most applications; therefore, it is recommended that whenever possible you use the AppleTalk library functions instead of the ioctl calls.

AppleTalk Link Access Protocol

The AppleTalk Link Access Protocol (ALAP), as described in *Inside Appletalk*, provides node-to-node "best effort" data delivery. The A/UX implementation of AppleTalk tries as well as possible to give the user direct access to this basic AppleTalk building block, but due to the relatively high level of the user interface, some restrictions are imposed. These restrictions include reduced throughput due to the overhead of system calls, some small speed penalty due to the layers being writting in C rather than 68k machine code, and some additional confusion due to the nature of the forwarder/STREAMS driver implementation of the interface.

Two special files provide access to the ALAP layer as do several AppleTalk library functions:

```
/dev/appletalk/lap/localtalk0/control
/dev/appletalk/lap/localtalk0/circuits
```

The ALAP implementation is multiplexing. This means that there is one connection downstream (the network), and multiple connections upstream. The connections upstream are viewed from the user interface as a "cloneable" file with the name `circuits`. Direct control of ALAP is done through the other entry point, `control`.

In addition, the file

```
/dev/appletalk/lap/localtalk0/.atnode
```

contains the last valid ALAP node number.

Note: Remember that ALAP is not accessed through the AppleTalk socket like DDP and the other higher level AppleTalk protocols. Also note that ALAP is part of this implementation's STREAMS driver, along with DDP, and does not require pushing or popping as would a module. Refer to "The AppleTalk Model" for more information.

The following streams I_STR ioctls, as defined in <at/alap.h>, are available at the ALAP layer:

AT_LAP_GET_CFG
: This ioctl gets the configuration table from the ALAP stream end. ic_dp should contain a pointer to a buffer of the type at_lap_cfg_t, as defined in <at/alap.h>, which it then fills with the table.

AT_LAP_LOOKUP
: This ioctl returns a table of registered ALAP types. ic_dp should contail a pointer to a buffer of the type at_lap_entry_t, (defined in <at/alap.h>).

AT_LAP_OFFLINE
: This ioctl takes the network offline; all local AppleTalk sockets become inoperative immediately. You must be the superuser to perform this function.

AT_LAP_ONLINE
: This ioctl brings the network online. All buffers and statistics are reset and the system's AppleTalk node ID is arbitrated for per the guidelines in *Inside Appletalk*. Note that the last valid ALAP node number is contained in the file /dev/appletalk/lap/localtalk0/.atnode. You must be the superuser to perform this function.

AT_LAP_REGISTER
: This ioctl registers an ALAP type. A pointer to the structure at_lap_entry_t must be passed in *arg*->ic_dp, with the type and name filled in by the calling application. Upon return, the circuit field will contain the virtual circuit on which that type is registered. Note that the final close will deregister the ALAP type.

  Note: The only ALAP types supported in the current distribution are DDP short and DDP long.

AT_LAP_SET_CFG
: This ioctl sets the configuration table in the ALAP stream end. You must pass an *arg*->ic_dp a pointer to a buffer of the type at_lap_cfg_t, as defined in <at/alap.h>. If they are nonzero, the following fields from the at_lap_cfg structure are copied into appropriate values in the ALAP stream end:

`initial_node`	the first ALAP node number to try on startup (All other fields in `at_lap_cfg` are ignored.)
`rts_attempts`	the total number of retry attempts

`AT_SYNC`
This ioctl blocks until the downstream and upstream queues are drained. It is useful for determining that a message has been sent. not only from node-to-node, but also from socket-to-socket and from internet-to-internet.

AppleTalk Datagram Delivery Protocol

The AppleTalk Datagram Delivery Protocol (DDP) extends ALAP's "best effort" to perform socket-to-socket delivery of datagrams over a LocalTalk internet. The A/UX implementation of AppleTalk tries as well as possible to give the user direct access to this low-level AppleTalk building block.

DDP uses AppleTalk socket as already described. Refer to the section "AppleTalk Sockets" for information on opening and cloning DDP AppleTalk sockets.

The following `I_STR` ioctl, defined in `<at/ddp.h>`, is available at the DDP layer:

`AT_DDP_GET_CFG`
This ioctl returns the DDP configuration information. `ic_dp` must contain a pointer to a buffer of the type `at_ddp_cfg_t`, as defined in `<at/ddp.h>`.

Reading DataGrams

The data read from the AppleTalk socket is defined and contained in the structure `at_ddp_t` (defined in `<at/ddp.h>`). `at_ddp_t` is an extended datagram (LAP type 2) without a LAP header. Short datagrams (LAP type 1) that are received are converted to the extended type for the user. Datagrams received that have no listener or have an error are thrown away. Error statistics are kept and can be accessed. (See `appletalk`(1M) in *A/UX System Administrator's Reference*, `lap_getinfo`(3N) in *A/UX Programmer's Reference*, and the ioctl call `AT_LAP_GET_CFG` in this manual.) A `read`(2) call will return the length of data read, from 13 bytes (minimum length of a DDP packet) to 599 bytes (maximum length of a DDP packet). Values outside this range are unreasonable.

If the DDP packet length is greater than the maximum length specified in the `read`(2) call, the bytes not read from the DDP packet will be silently (that is, with no indication of error) thrown away. Remember that `read` must be supplied with a count >= the largest number of characters expected to fill the DDP structure (i.e., 599 bytes). This is the default mode of the stream head for AppleTalk, with the `RMSGD` stream `read` option set. The `read` option may be changed by the user. See *UNIX System V STREAMS Programmers Guide* for more information on the use of STREAMS `read` options.

The `read` will always block until either a packet is received for the AppleTalk socket, an end-of-file condition occurs or an error condition occurs.

Reading Pending Datagrams

The length of the next (if any) datagram may be determined by using a standard ioctl call, as follows:

```
int  status;
int  nextdg_len;

status = ioctl(fd,FIONREAD, &nextdg_len);
```

where *fd* is the AppleTalk socket, `FIONREAD` is a constant defined in `<sys/ioctl.h>`; and `&nextdg_len` is a pointer to an integer variable in which the number of bytes of the next datagram to be read will be returned. If no datagram exists for the AppleTalk socket, a value of zero will be placed in `next_dg_len`.

Other A/UX calls you may find useful are `select`(2N), `FIOASYNC`, and `FIONBIO`. For more information on thees calls, see `termio`(7).

As always, upon successful completion of a `read`, the returned value indicates the number of bytes actually read. Upon error, a −1 is returned, and `errno` is set to indicate the error.

Writing Datagrams on a DDP Socket

The same parameters apply as for reading datagrams, namely, a buffer of the type `at_ddp_t`, (defined in `<at/ddp.h>`). should be written using `write`(1). Unlike reading datagrams, none of the standard ioctl calls are available.

When writing datagrams, the following fields in structure `at_ddp_t`, must be filled in by the user:

`dst_net`
: The current network number; if this is zero, the current network number will be filled in by DDP.

`dst_socket`
: The actual AppleTalk socket number; or, you may specify zero as this field and the number will be supplied.

`dst_node`
: The current node number; or, you may specify zero, as this field and the number will be supplied.

`checksum`
: If the DDP `checksum` field is zero, no checksum will be computed. If the field contains a nonzero value, a checksum will be computed and put in the field.

A `write`(2) call must write data from 13 bytes (minimum length of a DDP packet) to 599 bytes (maximum length of a DDP packet). Values outside this range are unreasonable.

As always, upon successful completion of a `write`, the returned value indicates the number of bytes actually written. Upon error, a –1 is returned, and `errno` is set to indicate the error.

Name Binding Protocol

The Name Binding Protocol (NBP) is a STREAMS module that converts the name of a network entity into a AppleTalk internet address. By "name" we mean an NBP tuple (see "NBP Packet Formats" in *Inside AppleTalk* for details) where the complete internet address is given in the form *<obj_len>object<type_len>type<zone_len>zone*.

In AppleTalk, names are dynamic; this means that a service can change locations and the clients can follow, as opposed to having fixed I/O resources or devices assigned to specific clients or services. The network itself still needs a few fixed services (such as the NBP, one of the static AppleTalk sockets that function like a post office) to be able to find anyone.

As with the other modules, it is necessary to push module `at_nbp` into the opened stream before starting requests.

Note: The NBPD daemon module `at_nbpd` and its associated daemon, `/etc/at_nbpd`, are already loaded at AppleTalk startup time. See `appletalk`(1M) for more

information.

The following I_STR ioctls, defined in <at/nbp.h>, are available at the NBP layer.

AT_NBP_LOOKUP
: This ioctl looks up services, returning the names of all entities matching the request, along the their network addresses. ic_dp should contain a pointer to a buffer large enough to hold an array of type at_nve (defined in <at/nbp.h>) structures, at least NBP_NAME_MAX characters in size.

AT_NBP_CONFIRM
: This ioctl confirms the name and address supplied as that of the service indicated. The data structure returned to the buffer pointed to by ic_dp is an array of type at_nve (defined in <at/nbp.h>).

AT_NBP_REGISTER
: This ioctl registers a name on an AppleTalk socket. The data structure returned to the buffer pointer to by ic_dp is an array of type at_nve (defined in <at/nbp.h>) containing all pertinent information registered with the local AppleTalk NBP daemon, at_nbpd.

AT_NBP_LOOK_LOCAL
: This ioctl looks up a service locally (that is, with the local NBP daemon at_nbpd), and returns into the buffer pointed to by ic_dp an array of type at_nve (defined in <at/nbp.h>).

AT_NBP_DELETE
: This ioctl cancels registration for the AppleTalk socket used for the call. Requires no passing of parameters.

AT_NBP_DELETE_NAME
: This ioctl cancels registration for a named AppleTalk socket. ic_dp must point to a buffer that contains an AppleTalk tuple name.

AT_NBP_SHUTDOWN
: This ioctl shuts down network registration. Requires superuser permission for access.

AppleTalk Transaction Protocol

The AppleTalk Transaction Protocol (ATP) is a STREAMS module that provides reliable transport of client data packets from a source AppleTalk socket to a destination AppleTalk socket.

A ATP connection can be opened with one of the ATP AppleTalk library routines, or in the manner described in the previous section, "AppleTalk Sockets." As with the other modules, it is necessary to push module `at_atp` into the opened stream before starting requests. This is done for the user if the AppleTalk library ATP `open` function is used (see `atp`(3N)).

The ATP implementation of AppleTalk provides two interface formats, synchronous and asynchronous. The synchronous interface is much simpler to use than the asynchronous interface and is suitable for most applications.

In A/UX, only one ioctl call can be pending on a stream at any one time. Synchronous calls wait; asynchronous calls do not wait.

If you wish to allow more than one process access to the AppleTalk socket, or wish for a process to have more than one transaction in progress at the same time, you must use the asynchronous interface.

The following table shows possible ATP request-response type combinations. The combination shown in italics (asynchronous/synchronous) is most common, as it allows the server to await requests, while the client can send requests immediately.

Server	**Client**
synchronous	synchronous
asynchronous	*synchronous*
synchronous	asynchronous
asynchronous	asynchronous

ATP packets must be preceded by both an ATP header and a DDP extended header (A LAP header is also appended, but this is done automatically as part of the module communication between DDP and LAP.) Packets being sent must have appropriate parts of these headers filled in. Other parts of the packets (such as the DDP source address) are filled in by ATP, DDP, or ALAP as part of the streams module communications as the packet is sent downstream. In particular, the following fields must be filled in by the user:

The variables in the DDP Header are:

`dst_net`
`dst_node`
`dst_socket`
: These three fields (part of structure `at_ddp_t` as defined in `<at/ddp.h>`) identify the remote AppleTalk internet address with which the transaction is being exchanged.

  When originating, these variables should be filled in with values returned from a previous NBP lookup request or other broadcast function.

  In replying to a previous message, these variables should be filled in with the source internet address values contained in the requesting message.

The variables in the ATP Header, which are are part of structure `at_atp` as defined in `<at/atp.h>`, are:

`at_atp_xo`
: This is set to nonzero when making a transaction request to indicate that "exactly-once" service is required; see *Inside AppleTalk*.

`at_atp_bitmap_seqno`
: On a request, this mask is used to indicate the number of packets to be returned. Starting with bit 0, a bit is set for each packet that can be received (up to a maximum of 8 packets). When you receive such a request, respond only with the packets corresponding to the bits set. (Only in execute-at-least-once mode does this packet differ from what the requester sent). When replying to a request, this field contains the reply ID number (0-7), which identifies the packet's order in the response; see *Inside AppleTalk*.

`at_atp_transaction_id`
: When originating, this variable should be incremented to insure a unique number for each new ATP transaction.

  When replying, this field should be the same as that in the request to which the reply is being made. This, along with the DDP information, identifies the transaction to which you are replying; see *Inside AppleTalk*.

Use the following macros (defined in `<at/atp.h>`) to help access the headers of ATP packets:

AT_ATP_HDR_SIZE
A constant that contains the size of the required ATP/DDP header.

AT_ATP_DATA_SIZE
The total maximum size of an ATP packet (this is the size you should use when allocating buffers, in particular those used when getting requests).

Note: The at_ddp_t structure contains type at_atp (also a structure) as a subpart in the data field. These defines set default values for most fields. If you need to set your own default values (via an AT_ATP_SET_DEFAULT), a structure of type at_atp_set_defaults will be prefixed to the at_ddp_t structure.

ATP Synchronous ioctl Calls

The following ioctls can be used to perform synchronous ATP functions. Make sure that the DDP AppleTalk socket address and ATP transaction ID are filled in in the buffer before use. Each ioctl has corresponding library routines that can be called instead.

AT_ATP_GET_REQUEST
This ioctl blocks until an incoming request arrives. The request is copied into the buffer pointed to by ic_dp and will be a structure of type at_ddp_t (defined in <at/ddp.h>).

AT_ATP_SEND_RESPONSE
This ioctl sends a response message to a remote ATP/DDP socket (specified in the message's header) in response to a message received via an AT_ATP_GET_REQUEST ioctl. The at_atp_bitmap_seqno field (defined in <at/atp.h>) must be filled in with the packet's response index. (See *Inside AppleTalk*.) It responds with a structure of type at_ddp_t (defined in <at/atp.h>) pointed to by ic_dp.

AT_ATP_SEND_RESPONSE_EOF
This ioctl is the same as the one above, except that it denotes the last message of a response.

AT_ATP_RELEASE_RESPONSE
This ioctl releases (cancels) a pending response. It requires

that the DDP/ATP header (a structure of type `at_ddp_t` that would normally be passed as part of a response) be sent with it.

`AT_ATP_ISSUE_REQUEST`
`AT_ATP_ISSUE_REQUEST_DEF`
Both of these ioctls send the ATP packet, a structure of type `at_ddp_t` pointed to by ic_dp, as a request to a remote AppleTalk socket. The `at_atp_xo` field (defined in `<at/atp.h>`) in the ATP header should be set to indicate whether or not execute-only-once mode is to be used for this transaction. You must fill in the header's `at_atp_bitmap_seqnofield` (defined in `<at/atp.h>`) with a bitmap corresponding to the number of packets expected to be returned. The requesting buffer is replaced by the response. If there is more than one response message, the rest of the messages are appended, in order, after the header. Only one copy of the header is returned, of a structure type `atp_result` (defined in `<at/atp.h>`). The `AT_ATP_ISSUE_REQUEST_DEF` ioctl requires an `atp_set_default` structure to be prefixed to the ATP packet buffer, and thus sets up the user's prefered defaults.

ATP Asynchronous `ioctl` Calls

Most asynchronous calls, such as responses to incoming requests, are nonblocking.

Normally the server side of an asynchronous transaction includes getting requests and polling to see if the request is complete. The client is notified when the transaction is complete.

Two synchronous calls, however, do perform blocking. The two asynchronous transactions that do block are `AT_ATP_ISSUE_REQUEST` which issues a request to a remote system and `AT_ATP_GET_REQUEST` which waits for incoming requests. Both ioctl's will block until the request completes or a time out occurs.

The basic asynchronous mechanism is to issue a command (such as a request) using the variant "no wait" form (ioctl: *cmd*`_NW`, where *cmd* is the preceding ioctl name), which will return immediately. At a later time, the user then issues a poll ioctl (`AT_ATP_POLL_REQUEST`) to see if the action has completed.

A variant of the basic asynchronous transaction begins with the issue of a "note" request. The "note" request asks the AST streams module to send a single byte to the calling AppleTalk socket when the action is complete. This byte must be read before making the poll call to get the results. Failure to do so can cause eronious data to be returned.

This technique can also allow a user to use the `select`(2N) system call to wait for the completion of several file system actions while waiting for pending AppleTalk transactions to complete.

Another variant of the "issue request" call, the "*cmd*`_DEF`" (short for "default") versions take the same arguments, except that the ATP packet passed to the ioctl must have space for a `atp_set_default` data structure in front of it. Two additional parameters are passed in the `atp_set_default` data structure, the rate and the number of retries.

The following are the "normal" nonblocking asynchronous ioctls. Their calling parameters are the same as those just described. They all return their transaction ID as an integer, which should be used to identify the transaction when polling for the it's completion. You may have many pending transactions outstanding at any one time.

`AT_ATP_ISSUE_REQUEST_NW`
: The three "no wait" requests return immediately. They allow a zero byte to be read from the AppleTalk socket (via a poll; see `AT_ATP_POLL REQUEST`, below) later, when the transaction is complete.

`AT_ATP_ISSUE_REQUEST_NOTE`
: The "note" requests are the same as the "no wait" requests, except that a single byte can be read from the AppleTalk socket when the transaction is complete. This byte contains a binary 1, to distinguish it from notes from incoming transactions.

`AT_ATP_POLL_REQUEST`
: Poll requests use the transaction ID number returned when a nonblocking request was made to poll to see if the transaction has completed and if a response is available. If it returns 0, the transaction has succeeded and the responses have been returned. If the transaction has not completed, it will return −1 with the error code `EAGAIN`. Otherwise, it completed with error and returned an appropriate error code in `errno`.

AT_ATP_GET_REQUEST_NW
AT_ATP_GET_REQUEST_NOTE
When you wish to connect with an incoming request asynchronously, you must first notify ATP that you wish to receive incoming requests. You may have more than one request outstanding at any one time (this is a good idea, in order not to miss incoming requests).

These two ioctls are defined to notify ATP that it may receive incoming requests. Note that they do *not* return a transaction ID as do the "issue request" calls above. These two variants on the synchronous form, "no wait" and "note," work in a similar manner to the "no wait" and "note" calls described above, except that, when a note call completes, a zero byte can be read at the input.

AT_ATP_GET_POLL
The "get next poll" call polls from the responding side to see if an ATP request has arrived. If so, it returns 0; otherwise, it returns –1 with the error EAGAIN.

Issuing ATP Requests

The response message from an ATP request has a header prefixed to it. This header, described by a structure atp_result (defined in <at/atp.h>), provides information about the number of response packets returned, the offset to the DDP header returned, the offsets to the ATP headers & the data that follow them for each of the responses, and the lengths of the response packets (including ATP headers).

All offsets and counts must be given in bytes and the offsets are absolute offsets from the beginning of the ATP packet.

AT_ATP_SET_DEFAULT
When issuing an ATP request, you can specify your own values for the number of retries and the rate of retries of the transaction on the network. If you do not specify them, default values are used. These defaults, attributes of an open ATP socket, can be set and changed with a call to AT_ATP_SET_DEFAULT. The default values will be changed until either a close or the next AT_ATP_SET_DEFAULT is received.

The data structure `atp_set_default` has two fields:

`def_rate`
: is the retry rate in hundredths of a second. The internal ATP timer may not be able to resolve timeouts this small. If it cannot resolve the timeout, it makes a best effort instead.

`def_retries`
: is the number of times to retry the transaction before giving up. The value `ATP_INFINITE_RETRIES` can be used to say "retry forever."

Printer Access Protocol

The Printer Access Protocol (PAP) is designed primarily to send data to printers and print servers. It is constructed in a server/client relationship, where the server is the printer or print server and the client is the application that wants to do the printing.

For a client process, the AppleTalk socket can be opened with one of the AppleTalk library functions (such as `at_pap_open_nve`(3N)) or in the manner described in the section on AppleTalk socket.

> *Note:* remember that if a direct open of the AppleTalk socket is done, you must push the ATP (`at_atp`) and PAP (`at_pap`) modules into the stream *in that order.* Failure to do so can cause unpredictable results.
>
> Also note that the AppleTalk library function `at_pap_open_nve`(3N) does this for the user and in the correct order. (see `pap`(3N) for details)

The same proceadure is used to turn an AppleTalk socket into a server process, only the other PAP module, `at_papd`, is used.

The following stream `I_STR` ioctls, defined in `<at/pap.h>`, are available at the PAP level:

`AT_PAPD_GET_NEXT_JOB`
: This ioctl call is made by a server; that is, a stream that has the `at_papd` module pushed on, when it is ready to respond to a new PAP client. You pass a pointer to structure `at_ddp_t` (datagram packet) in `ic_dp`. `at_ddp_t` contains the structure `at_atp` (AppleTalk transaction packet) with an `atp_set_default` structure prefixed to it. It returns the same structure with the destination fields filled in by the client that has been waiting the longest for service.

The returned structure is suitable to be reused in a AT_PAP_SETHDR ioctl command used to inform the PAP client that you are accepting the job.

AT_PAP_SETHDR
This ioctl sets the fields in the PAP header so they do not have to be set on each ioctl call for a given transaction. The following fields should be filled in with this call:

```
ddp = ATP_DDP_HDR(&buf[sizeof(struct atp_set_default)]);
            /* this will always be a PAP request */
ddp->type = 3;
atp = ATP_ATP_HDR(&buf[sizeof(struct atp_set_default)]);
            /* 1 if exactly once, otherwise 0 */
atp->at_atp_xo = X;
            /* set true only on last message */
atp->at_atp_eom = 0;
            /* send transaction status, normally 0 */
atp->at_atp_sts = 0;
            /* always 0 */
atp->at_atp_unused = 0;
            /* Should be set 0 */
atp->at_atp_user_bytes[0] = 0;
            /* and reset after use */
atp->at_atp_user_bytes[1] = 0;
atp->at_atp_user_bytes[2] = 0;
atp->at_atp_user_bytes[3] = 0;
```

If the destination fields are filled in, the connection ID must be placed in `at_atp_user_bytes[0]`.

AT_PAP_SET_STATUS
This ioctl changes the status string associated with a PAP server AppleTalk socket. This is the string returned to a PAP client in a PAP open connection reply or status reply packet. `ic_dp` points to a non-null-terminated string and `ic_length` contains the length of that string. Strings longer than 255 characters are truncated.

AT_PAP_READ
This ioctl reads data from a client PAP AppleTalk socket and places up to 512 characters into the buffer pointed to by `ic_dp`. Data are read from one ATP response packet at a time, and any data left in an ATP packet after `ic_len` bytes are copied are lost. A value of −1 is returned if the

connection to the other end has been broken.

AT_PAP_READ_IGNORE
This ioctl requests a `read` from the other side but to throw data away if the data actually arrives. This is useful for printing to a LaserWriter.

Note: LaserWriters require that a client read request be pending at all times in order to take LaserWriter generated status/error messages. Because STREAMS ioctl calls are synchronous A/UX AppleTalk cannot read and write at the same time. Therefore, we always have a `read` request pending for which we discard the results.

AT_PAP_WRITE
This ioctl sends `ic_length` bytes pointed to by `ic_dp` to the other end of the current PAP connection. `ic_length` cannot exceed 512 bytes.

Note: Another LaserWriter idiosyncracy is that all packets (except the last end-of-file packet) **must** have exactly 512 bytes of PAP data. Failure to do so can cause the print job to be ignored or cause the printer to hang up.

AT_PAP_WRITE_EOF
This ioctl sends `ic_length` bytes (pointed to by `ic_dp`) to the other end of the PAP session. `ic_length` cannot exceed 512 `mefsmes` (bytes). It also sends a PAP end-of-file indication to the other end to indicate that no more data will be sent. It does an implicit `AT_PAP_WRITE_FLUSH`.

AT_PAP_WRITE_FLUSH
This ioctl sends `ic_length` bytes (pointed to by `ic_dp`) to the other end of the PAP session. `ic_length` cannot exceed 512 (bytes). Since PAP runs on top of ATP, PAP `writes` are queued up on the receive side until either a complete ATP response is available (up to 4K bytes) or an ATP end of message is sent. This call sends an ATP end of message, which causes all waiting PAP writes to be read by the PAP module on the other end. This should be done if a higher level protocol (for example, a handshake with a LaserWriter) needs to do a `write` followed by a `read`.

MACHINE DEPENDENCIES

Some computers (notably the MC68000 processor used on the AST-ICP) require certain data types to be aligned on memory boundaries. Because of variables (such as header length) used in the AppleTalk protocols, alignment on address boundaries cannot be guaranteed, and access to short and long fields must be done in special ways.

For example, the `at_net` type represents the internetwork net identifier. This identifier occupies 2 bytes. The ALAP header is 3 bytes long and it is reasonable to expect that if a ALAP frame is read into an even-aligned buffer, the DDP component will start on an odd boundary. A short read of the `net_id` would then result in the generation of an address error by a 68000-family processor. The solution is to use routines that read and write by bytes.

Packet Size Limitations

Packet size limitations are as follows:

Protocol	Packet data size (bytes)
ALAP	600
DDP	586
ATP (response)	8 packets of 578 (see below)
PAP	512
NBP	8192

where the ATP response packet data size is 582 bytes if the ATP user bytes are filled.

Signed Versus Unsigned

The difference between the C language types `signed` (the default) and `unsigned` is that signed types can represent negative numbers. For example, the program

```
{
    char                            c = 254;
    unsigned char                   uc = 254;

    printf("char = %d, unsigned char = %d0, c, uc);
}
```

will print out

```
char = -2, unsigned char = 254
```

So, you can see that you have to be careful. Signed types will do two things: sign-extend and one's-fill. Sign extension occurs when ones are added at the beginning of a variable to keep it a

negative number. For example, in the following:

```
ui = c
```

where `c` is from the program above, and `ui` is an `unsigned int`, `ui` will be equal to `0xfffe`. One's-fill occurs when a variable is shifted right, that is, >>. In signed quantities, whatever is the Most Significant Bit will be duplicated and made the new MSB. With unsigned quantities, a shift right will fill the vacated MSB with zeros.

When dealing with fields that are packed into communication packets, (such as the ATP bitmap) you must be careful not to change the values unexpectedly when changing from type to type.

Bit Manipulation

Another little-used programming area is bit manipulation. This is particularly valuable when dealing with the ATP bitmap/sequence field. To test if the fifth response came back, you might use:

```
if (bitmap & 1<<4)
{
}
```

Remember that bits are numbered right-to-left 0 ... 32. To turn on bit 2, you might use:

```
bitmap |= 1<<2;
```

Or, to turn off bit 2, you might use:

```
bitmap &= ~(1<<2);
```

ERRORS

The following error codes pertain to all (or most) layers. Errors lag behind the event; an error caused by a `write` will not be seen at the completion of that `write`, but will be seen on some subsequent access to the stream, usually the next one. Once a stream is in an error condition, it behaves very poorly, and only a `close` will reset it. This holds for those layers that are part of the stream itself (ALAP and DDP).

Most PAP and ATP errors do not have such dire consequences. These error conditions are passed back as a field in a structure.

Possible error messages from `read` or `write` commands used in any of the protocol layers are as follows:

`[EACCES]`	You tried to open a static socket and you are not a superuser.

[EBADF]	fd is not a valid file descriptor open for read/write
[EFAULT]	buf points outside the allocated address space.
[ERANGE]	Message size outside legal range.
[ENOSPC]	Downstream write queue is full, and the O_NDELAY flag is set.
[EINTR]	A signal was caught while waiting for downstream queue space or a message buffer.
[ENXIO]	An M_HANGUP message was received at the stream head; the minor number is greater than the number of AppleTalk sockets for this network.
[EBADMSG]	Message waiting to be read is not of type M_DATA.
[ENODATA]	No message waiting to be read, and O_NDELAY set.
[ENETDOWN]	The network is down, and cannot receive data.
[ENOTCONN]	The module is not connected. It is likely that a close connection request was received from the other side.
[ENOTSOCK]	You specified an invalid dst_socket.
[EADDRNOTAVAIL]	You specified an invalid dst_node.
[EMSGSIZE]	Your message length exceeded the limits of a DDP DataGram.
[ERANGE]	The wrtoffset you specified was too small.
[ENETDOWN]	The net is down.
[ENOBUFS]	No buffers are available to hold your message.

For other errors, see ioctl(2), open(2), write(2), and read(2).

WARNING

The AppleTalk library routines automatically set up and invoke the correct ioctl requests. The ioctls give the programmer more control, but they require a much greater understanding of the A/UX AppleTalk software. In addition, AppleTalk ioctl calls are subject to change, while AppleTalk library functions will not change. It is, therefore, strongly recommended that the library routines be used whenever possible instead of the more complicated ioctl calls.

FILES

```
/dev/appletalk/socket
/dev/appletalk/socket1
/dev/appletalk/socket2
     ...     ...
     ...     ...
/dev/appletalk/socket127
/etc/at_nbpd
/dev/appletalk/lap/localtalk0/control
/dev/appletalk/lap/localtalk0/circuits
/dev/appletalk/lap/localtalk0/.atnode
```

SEE ALSO

appletalk(1M), creat(2), dup(2), fcntl(2), ioctl(2), open(2), at_ident(3N), atp(3N), ddp(3N), lap(3N), nbp(3N), pap(3N), zip(3N), forwarder(7); "AppleTalk Programming Guide" in *A/UX Network Applications Programming*; *Inside AppleTalk*; *UNIX System V STREAMS Programmer's Guide.*

NAME

forwarder — forwarder device driver

DESCRIPTION

The forwarder is a specalized streams device driver written so as to be able to run on a wide range of Front End Processors (FEP).

> *Note:* Apple® currently only supports the forwarder software on the AST Intelligent Communications Processer (AST-ICP). It can, however, be ported to any number of other communications processers.

The FEP generally has a CPU, a memory, I/O circuitry devices, and a means of communicating with the host Macintosh® II via the NuBus™. (Modules are normally downloaded onto the FEP allowing for offloading of the host processer.)

The forwarder software is actually a twin situation; an identical copy is kept in the kernel on the host and in the minioperating system found on the FEP. The twins work together (as a matched pair) to pass messages and data across the NuBus. From the kernel it looks like a stream driver, from the actual stream driver (or modules) it looks like a stream head.

The forwarder software knows that there is a processing or space separation (the NuBus) between the operating system and the remote modules and streams driver. It is the only module that needs to know about this division of powers; it hides this fact from the other layers.

Because the NuBus exists, however, there are some stream restrictions of which the implementor must be aware. Any operation that transverses the forwarder must pass through the forwarder's queue processing. For example,

```
q->q_next->q_next
```

would be incorrect because it is trying to access the queue beyond the forwarder, and that is impossible. Careful thought and an understanding of the forwarder's task should help prevent such errors.

When it is next to a forwarder, the stream head behaves differently when it receives an `I_PUSH` ioctl. It first checks the module id number downstream. If the ID number is ≥ `FORWARDERMIN` but ≤ `FORWARDERMAX`, it sends an `I_PUSH` via an `M_IOCTL` message. The forwarder passes the request to its twin on the board, which tries to open the indicated module. The forwarder then responds with an "acknowledge" if the `open`

completed. If the open did not complete successfully, a "negative acknowledge" is returned. If the module is not found on the board, a message is returned to that effect and the stream head continues the push as if the forwarder were not there. The process is the same for popping, except there would be no "not found" case.

Control of the forwarder is done via stream I_STR ioctls. The following stream I_STR ioctls, defined in <fwd.h>, are available.

I_FWD_LOOKUP — Returns a table of the installed application strings and places it in the location pointed to by arg->ic_dp. I_FWD_LOOKUP returns a table into arg->ic_dp, where the line entries are of type struct fwd_entry, found in <fwd.h>. The length of the table is found in arg->ic_len, but is always less than the stream maximum of 1024 Kbytes.

I_FWD_RESET — An ioctl that resets the board into a state ready for downloading. This ioctl must be used when the system first comes up, or when an FEP panic occurs. A I_FWD_RESET will also disable any other applications currently talking to the board with EIO errors. Note that there are many examples of FEPs where software cannot issue a reset to the board. In this case, if the forwarder has lost communication with its twin, I_FWD_RESET will have no effect and you reboot the system to reset the forwarder.

I_FWD_DOWNLD — Causes the binary data contained in fwd_record.data to be downloaded to the FEP starting at FEP memory location fwd_record.begin. The structure fwd_record is defined in <fwd.h>.

I_FWD_UPLD — Causes the binary data to be uploaded from the FEP memory into the data field

fwd_record.data. fwd_record.ld_length is the number of bytes to be uploaded from the FEP. The structure fwd_record is defined in <fwd.h> .

I_FWD_START — Instructs the loader to transfer execution to the address contained in fwd_entry.start. The name field will be placed in the forwarder's application table.

EXAMPLE

```
int dev_fd;
struct strioctl i_str;

        if((dev_fd = open(dev_file, O_NDELAY)) < 0)
                HANDLE_ERROR();

        i_str.ic_cmd = I_FWD_DOWNLD;
        i_str.ic_timout = 4;
        i_str.ic_len = fwd_record.begin;
        i_str.ic_dp = fwd_record;

        if(ioctl(dev_fd, I_STR, &i_str) < 0)
                HANDLE_ERROR();
```

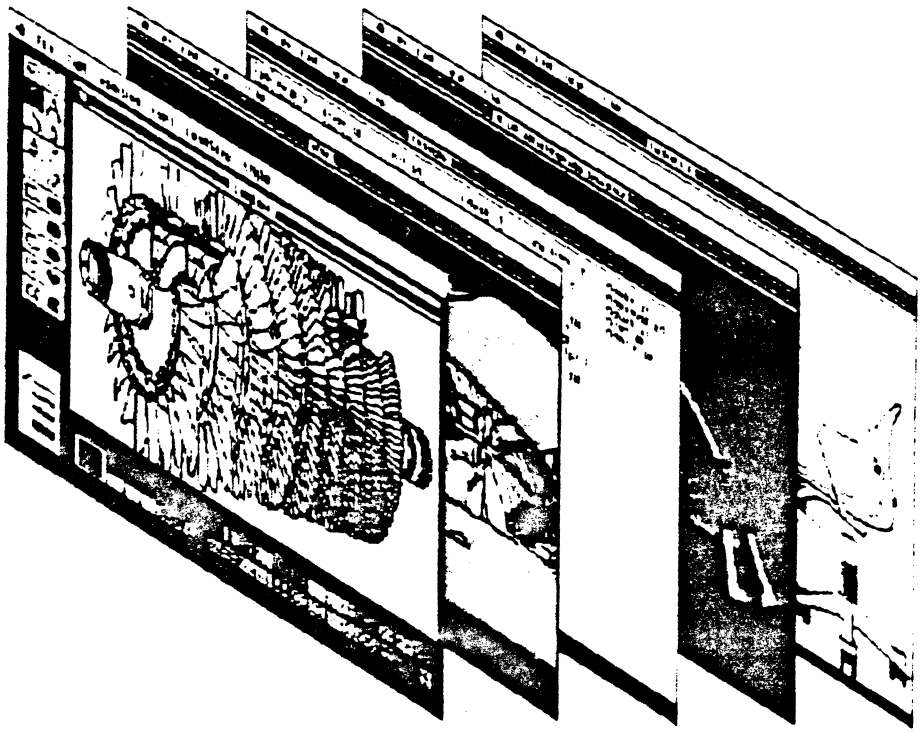
FILES

/dev/fwdicp11
/etc/startup.d/fwdicp.d/at_load
/etc/startup.d/fwdicp.d/tt_load

SEE ALSO

fwd_lkup(1M), fwdload(1M); *AT&T UNIX System V STREAMS Programming Guide.*

Apple® A/UX™ Command Reference



NAME

at_cho_prn — choose a user's default printer on the AppleTalk® network

SYNOPSIS

at_cho_prn [-t *type*] [-z *zone*]

DESCRIPTION

at_cho_prn interrogates the network to find out what printers have been registered on that network. By default, the command will display all entities of the types LaserWriter, ImageWriter, and AuxLpServer in the current zone.

at_cho_prn then prompts you to enter the number of the printer you want to select from the list it displays. It stores information about this printer in /usr/lib/PrinterChoices, keyed by your user-ID. This information is then used by at_printer(1) to determine your default printer.

EXAMPLE

The command:

```
at_cho_prn -t LaserWriter
```

would produce output such as:

```
ITEM OBJECT TYPE         ZONE  NET   NODE  SOCKET
1:   doc1   LaserWriter  *     5678  0xcb  0xaa
2:   doc2   LaserWriter  *     5678  0xd4  0xaa

P: ITEM number (0 to make no selection)?
```

where OBJECT is the name of the registered printer; TYPE is its type; ZONE is its zone name (where * designates the current zone); NET is its network number; NODE is its node number; and SOCKET is its AppleTalk socket number. The latter three numbers are in hexadecimal format.

FILES

/usr/bin/at_cho_prn
/usr/lib/PrinterChoices

SEE ALSO

at_nvelkup(1), at_printer(1), at_server(1), at_status(1); *Inside AppleTalk;* "Installing and Administering AppleTalk," in *A/UX Network System Administration*; "AppleTalk Programming Guide" in *A/UX Network Applications Programming*.

NAME

at_nvelkup — look up NVEs registered in the AppleTalk® network

SYNOPSIS

at_nvelkup [-l] [-t *type*] [-z *zone*] [-o *object*]

DESCRIPTION

at_nvelkup queries the NBP (Name Binding Protocol) module for the addresses of all NVEs (network visible entitites) registered on the AppleTalk zone. The default is to use the local zone (*) that matches the name specified by the user. The *object*, *type*, and *zone* of the NVEs may be specified to limit the lookup.

If the -l flag option is used, only the NVE's registered on the local node will be displayed.

Information about the NVEs is displayed in a table format, one line per NVE, containing the *object*, *type*, and *zone* names, and the network, node, and socket numbers in hexadecimal format, respectively. If the -l flag option is used, the ID of the process that registered the NVE and the time of registration will also be printed.

EXAMPLE

The command

```
at_nvelkup
```

will query the NBP daemon for all NVE's registered on the local AppleTalk zone. The following table is displayed:

```
OBJECT  TYPE          ZONE  NET   ND SK
doc1    LaserWriter    *    5587  c6 aa
doc2    LaserWriter    *    2222  d4 da
```

FILES

/usr/bin/at_nvelkup

SEE ALSO

at_cho_prn(1), at_printer(1), at_server(1), at_status(1); *Inside AppleTalk.*

NAME

at_printer — copy data to a remote PAP server

SYNOPSIS

at_printer [-o *object*] [-t *type*] [-z *zone*] [-u *uid*]

DESCRIPTION

at_printer opens a PAP (Printer Access Protocol) AppleTalk® connection to a remote PAP server, such as a LaserWriter®, and then copies its standard input to the remote server until it reaches an end-of-file. This command can be used to send "raw" PostScript™ to a LaserWriter (it will not engage in a dialog with a LaserWriter about fonts, or load PostScript header files in the same manner as a Macintosh™ Operating System does).

This command will check to see if the invoking user has previously "chosen" a default printer with the at_cho_prn(1) command. The name of the chosen server is stored on a per-user basis (by the user's UID) in the file /usr/lib/PrinterChoices.

A user may override the default *object*, *type*, and *zone* choices with the -o, -t, and -z flag options. If a user has not specified a printer choice with at_cho_prn, but explicitly specifies the *object*, *type*, or *zone*, these values will default to the values specified below.

at_printer takes the following parameters:

-o *object* The object name of the remote server. A list of available objects can be obtained using the at_nvelkup(1) command. If the object name is a wildcard (=), it will match the first available server with a matching type and zone. If this parameter is omitted, it defaults to =.

-t *type* The type of the remote server. If the type name is a wildcard (=), it will match the first available server with a matching *object* and *zone*. If omitted, this parameter defaults to LaserWriter.

-z *zone* This is the network zone in which the server exists. To access a server in the same zone as yourself you should use *. Wildcarding of zones is not allowed. Networks without bridges do not have zones and should always use the default *. If this parameter is omitted, it defaults to *.

-u *uid* This overrides a user's current user-ID when choosing a default printer from the file `/user/lib/PrinterChoices`. Typically, this flag option will be used by only `lp` interface programs.

`at_printer` will output a message designating to which server it is connected prior to transferring data.

EXAMPLE

```
at_printer -l < test.ps
```

will copy the PostScript file `test.ps` to the first available LaserWriter.

WARNING

`at_printer` does not attempt to interpret contents of input files. To print properly on a PostScript printer, ASCII files must be preprocessed through `pstext`(1) or `enscript`(1) and `troff`-formatted files must be preprocessed through `psdit`(1).

FILES

`/usr/bin/at_printer`
`/usr/lib/PrinterChoices`

SEE ALSO

`at_cho_prn`(1), `at_nvelkup`(1), `at_server`(1), `at_status`(1), `enscript`(1), `pstext`(1), `psdit`(1); *Inside AppleTalk; Inside Postscript/LaserWriter;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming*; "Installing and Administering AppleTalk," in *A/UX Network System Administration.*

NAME

at_server — a generic PAP server

SYNOPSIS

at_server -c *command* -o *object* [-t type]

DESCRIPTION

at_server is a simple generic PAP (Printer Access Protocol) server. It opens a PAP server AppleTalk® socket and registers itself on the local server with the name

object:*type*@*

When an incoming PAP request is received, at_server forks a process to read the data from the remote client and executes a command from the command parameter. Incoming data from the server is written to a pipe that can be read by the command as standard input. Note that the server is "one-way;" it only reads from the remote client. It does not engage in a dialogue with a client in the same manner that a LaserWriter does with the Macintosh Operating System.

The parameters that at_server takes are

command A shell command to be executed when an incoming connection is requested.

object The object name of the server; this is required and must not be a wildcard (=).

type The type name of the server (this is optional). If omitted, it defaults to LaserWriter. Again, wildcards are not permitted.

EXAMPLE

```
at_server -c 'lp -dziffel -s' -o piggie -t LaserWriter &
```

creates a PAP server of type LaserWriter called piggie that will accept incoming PAP requests from the network. Each one will have its data spooled locally by lp for printing on printer ziffel.

```
at_printer -o piggie -t LaserWriter < x.list
```

will send x.list to this server to be printed.

```
at_printer -o = -t LaserWriter < x.list
```

will send x.list to any available printer.

FILES

/usr/bin/at_server

SEE ALSO

at_cho_prn(1), at_nvelkup(1), at_printer(1), at_status(1); *Inside AppleTalk.*

NAME

at_status — return status from a PAP server

SYNOPSIS

at_status -o *object* [-t *type*] [-z *zone*]

DESCRIPTION

at_status gets the status string from an AppleTalk® PAP (Printer Access Protocol) server such as a LaserWriter®. It takes the parameters

object The object name of the PAP server. This parameter is required and wildcards are not permitted.

type Wildcards are not permitted; if this name is omitted, it defaults to LaserWriter.

zone The zone of the PAP server. Wildcards are not permitted; if *zone* is omitted, it defaults to *, your local zone.

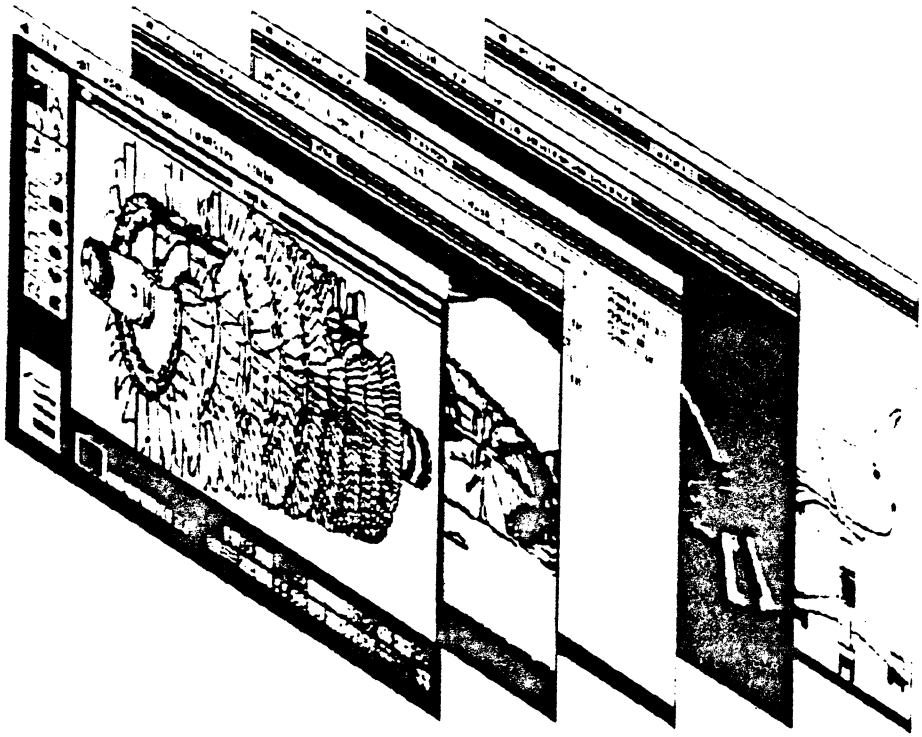
FILES

/usr/bin/at_status

SEE ALSO

at_cho_prn(1), at_nvelkup(1), at_printer(1), at_server(1); *Inside AppleTalk*.

Apple® A/UX™ Programmer's Reference



NAME

atp_open, atp_close, atp_sendreq, atp_getreq, atp_sendrsp — AppleTalk® Transaction Protocol (ATP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/atp.h>

int atp_open(socket)
at_socket *socket;

int atp_close(fd)
int fd;

int atp_sendreq(fd, dest, buf, len, userdata, xo,
            tid, resp, retry, comp, param);
int fd;
at_inet_t *dest;
char *buf;
int len, userdata, xo;
u_short *tid;
at_resp_t *resp;
at_retry_t * retry;
int (*comp)(), param;

int atp_getreq(fd, src, buf, len, userdata,
           xo, tid, bitmap, comp, param);
int *fd
at_inet_t *src;
char *buf;
int *len, *userdata, *xo;
u_short *tid;
u_char bitmap;
int *nresp, (*comp0(), param;

int atp_sendrsp(fd, dest, xo, tid,
            resp, comp, param);
int fd;
at_inet_t *dest;
int xo;
u_short tid;
at_resp_t *resp;
int (*comp)(), param;
```

DESCRIPTION

The ATP interface provides applications with access to the services of the AppleTalk Transaction Protocol.

These routines use the following structures, defined in <at/appletalk.h>.

at_inet_t specifies the AppleTalk internet address of a DDP AppleTalk socket endpoint.

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

at_retry_t specifies the retry interval and maximum count for a transaction.

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

The members of this structure are

interval The interval in seconds before ATP retries a request.

retries The maximum number of retries for this ATP request. If *retries* is AT_INF_RETRY, the request will be repeated infinitely.

at_resp_t, defined in <at/atp.h>, specifies buffers to be used for response data.

```
typedef struct  at_resp {
        int               count;
        struct iovec      resp[AT_ATP_TRESP_MAX];
        int               userdata[AT_ATP_TRESP_MAX];
} at_resp_t;
```

The members of this structure are

count The maximum number of responses expected (and for which buffers are allocated).

resp An iovec structure describing the response buffers and their lengths.

userdata An array of 32-bit words holding the user bytes for each ATP response.

`atp_open` opens an ATP AppleTalk socket and returns a file descriptor for use with the remaining ATP calls.

socket A pointer to the static DDP AppleTalk socket number to open. If the socket number is zero, a socket is dynamically assigned, and the socket number is returned in *socket*.

`atp_close` closes the ATP AppleTalk socket identified by the file descriptor *fd*.

`atp_sendreq` sends an ATP request to another socket. In synchronous mode, this call blocks until a response is received.

fd The ATP file descriptor to use in sending the request.

dest The AppleTalk internet address of the AppleTalk socket to which the request should be sent.

buf Specifies the request data buffer.

len Specifies the size of request data buffer size.

userdata Contains the user bytes for the ATP request header. This is the user-supplied data to be passed by the ATP request and will be a PAP (Printer Access Protocol) packet or other user-supplied data.

xo Should be true (1) if the request is to be an exactly-once (XO) transaction.

tid On return, contains the transaction identifier for this transaction. *tid* can be NULL if the caller is not interested in the transaction identifier.

`atp_sendreq` requires a pointer to an `at_resp_t` structure containing two arrays for the response data: *resp*, an eight-entry `iovec` array, and *userdata*, an eight-entry array. The field *iov_base* in each `iovec` entry, points to a buffer to contain response data. The field *iov_len* specifies the length of the buffer. The field *bitmap* indicates the responses expected; on return, it indicates the responses received. On return, each *iov_len* entry indicates the length of the actual response data. If the number of responses is less than expected, either an EOM was received or the retry count was exceeded. In the latter case an error is returned. Each *userdata* entry in *resp* contains the user data for the respective ATP response packet. The *retry* pointer specifies the ATP request retry timeout in seconds and the maximum retry count. If *retry* is NULL, the default timeout, `AT_ATP_DEF_INTERVAL`, and the default retries,

AT_ATP_DEF_RETRIES, are used. The *retries* field of *retry* can be set to AT_INF_RETRY, in which case the transaction will be repeated infinitely.

atp_getreq receives at ATP request sent from another AppleTalk socket. It completes when a request is received.

fd The ATP file descriptor to use in receiving the request.

src The AppleTalk internet address of the AppleTalk socket from which the request was sent.

buf Specifies the data buffer in which to store the incoming request.

len Specifies the data buffer size in which to store the the incoming request.

userdata On return, contains the user bytes from the ATP request header. *userdata* can be NULL if the caller is not interested in the *userdata*.

xo Will be true (1) if the request is to be an XO transaction.

tid Contains the transaction identifier for this transaction.

bitmap Indicates the responses expected by the requester.

xo, *tid*, and *bitmap* are always returned, since the transaction may require a response.

atp_sendrsp sends an ATP response to another AppleTalk socket. All response data is passed in one at_sendrsp call. In the case of an XO transaction, the call does not return until a release is received from the requester, or the release timer expires. In the latter case, an error is returned.

fd The ATP file descriptor to use in sending the response.

dest The AppleTalk internet address of the AppleTalk socket to which the response should be sent.

tid Contains the transaction identifier for this transaction.

atp_sendrsp requires a pointer to an at_resp_t structure containing two arrays for the response data: *resp*, an eight-entry iovec array, and *userdata*, an eight-entry array. The field *iov_base* in each iovec entry points to a buffer containing response data. The field *iov_len* specifies the length of the

response data. Each *userdata* entry in *resp* contains the user data to be sent with the respective ATP response packet. The field *bitmap* indicates the responses to be sent.

ERRORS

All routines return -1 on error with a detailed error code in `errno`. For additonal errors returned by the underlying DDP and ALAP modules, see `ddp`(3N) and `lap`(3N).

`[EBADF]`	*fd* is not a valid file descriptor (all).
`[ENOTTY]`	*fd* is not a TTY, that is, not a special device (all).
`[EINTR]`	The request was interrupted by signal (all).
`[EAGAIN]`	The request failed due to a temporary resource limitation; try again. An XO transaction will not have been initiated if this error occurs (all).
`[EINVAL]`	Invalid *dest*, *len*, *resp*, or *retry* parameter (`atp_sendreq`). Invalid *len* parameter (`atp_getreq`). Invalid *dest* or *resp* parameter (`atp_sendrsp`).
`[ENOENT]`	An attempt to send a response to a nonexistent transaction (`atp_sendrsp`).
`[ETIMEDOUT]`	The request exceeded the maximum retry count (`atp_sendreq`).
`[EMSGSIZE]`	The response is larger than the buffer, or more responses were received than expected. Truncated to available buffer space (`atp_sendreq`). The request buffer is too small for request data, truncated (`atp_getreq`). The response is too large; maximum is `AT_ATP_DATA_SIZE` bytes (`atp_sendrsp`).

WARNINGS

The parameters *comp* and *param* allow asynchronous sending and receiving of ATP requests. At this release, asynchronous requests are not supported, and these parameters should be set to NULL to indicate synchronous operation.

The length of each response buffer, specified in *iov_len*, is overwritten by the actual response length when `atp_sendreq` returns.

SEE ALSO

`ddp`(3N), `lap`(3N), `nbp`(3N), `pap`(3N), `rtmp`(3N), `appletalk`(7); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming*.

NAME

ddp_open, ddp_close — AppleTalk® Datagram Delivery Protocol (DDP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/ddp.h>

int ddp_open(socket)
at_socket *socket;

int ddp_close(fd)
int fd;
```

DESCRIPTION

The DDP interface, when included in applications, provides access to the AppleTalk Datagram Delivery Protocol operations.

ddp_open opens a static or dynamic DDP AppleTalk socket and returns a DDP AppleTalk socket file descriptor which can be used to read and write DDP datagrams.

socket A pointer ordering the DDP AppleTalk socket number to open. If the AppleTalk socket number is 0, a DDP AppleTalk socket is dynamically assigned, and the socket number is returned in *socket*.

An error condition will result if there are no more dynamic AppleTalk sockets available, if the maximum number of open files has been exceeded at a process or system level, or if the network is offline.

Only the superuser can open a static DDP AppleTalk socket.

ddp_close closes the DDP AppleTalk socket identified by the file descriptor *fd*.

Datagrams are sent and received with the long DDP header format, using standard A/UX read(2) and write(2) system calls. If the datagram is directed to a LocalTalk™ interface on the same network, the DDP protocol module will send it with a short DDP header. The long header DDP datagram is defined by the following structure in <at/ddp.h>.

```
typedef struct {
        at_length   length;
        at_chksum   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;
        at_type     type;
        u_char      data[AT_DDP_DATA_SIZE];
} at_ddp_t;
```

When writing a datagram, only the fields `checksum`, `dst_net`, `dst_node`, `dst_socket`, `type`, and `data` need to be set. `length` is the DDP packet length and hop count field. The hop count is in the 6 most significant bits of this field; the length is in the 10 least significant bits.

`checksum` contains the DDP checksum. When writing datagrams, a checksum is only computed if this field is nonzero.

Datagrams can be sent and received asynchronously using standard A/UX facilities: `select`(2N); `O_NDELAY` `fcntl`(2); or `FIONREAD`, `FIONBIO`, and `FIOASYNC` ioctls (see `ioctl`(2)).

ERRORS

All routines return -1 on error with detailed error code in `errno`:

`[EBADF]`	*fd* is not a valid file descriptor (all).
`[ENOTTY]`	*fd* is not a TTY, that is, not a special device (all).
`[EINTR]`	The request was interrupted by signal (all).
`[EAGAIN]`	The request failed due to a temporary resource limitation; try again (all).
`[EACCES]`	A nonsuperuser attempt to open a static AppleTalk socket (`ddp_open`).
`[EINVAL]`	An attempt is made to open an invalid AppleTalk socket number (`ddp_open`).
`[EADDRINUSE]`	The static socket is in use, or all dynamic sockets are in use (`ddp_open`).
`[EADDRNOTAVAIL]`	A write is attempted to an invalid node number.
`[EMSGSIZE]`	A datagram is too large.
`[ENETDOWN]`	The network interface is down (all).

[ENXIO]	Out of file descriptors (all).
[EWOULDBLOCK]	Asynchronous read or write would block, except for read with O_NDELAY would block
[ENODATA]	Asynchronous read with O_NDELAY would block.

Routines also return any additional error codes returned by the underlying ALAP module (see lap(3N)) and by standard A/UX open(2), close(2), read(2), write(2), and ioctl(2) system calls.

FILES

/dev/appletalk/ddp/*
/etc/appletalk

SEE ALSO

close(2), fcntl(2), ioctl(2), open(2), read(2), select(2N), write(2), atp(3N), ddp(3N), lap(3N), nbp(3N), pap(3N), rmtp(3N), fcntl(5), termio(7), appletalk(7); *Inside AppleTalk;* "AppleTalk Programming Guide" in *A/UX Network Applications Programming*.

NAME

lap_bind, lap_close, lap_default, lap_getinfo, lap_init, lap_name, lap_open, lap_setinfo, lap_shutdown — AppleTalk® Link Access Protocol (ALAP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/alap.h>
#include <at/lap.h>

int lap_bind(if_id, type, name)
int if_id;
short type;
char *name;

int lap_close(if_id)
int if_id;

char *lap_default()

int lap_getinfo(if_id, info)
int if_id;
at_ifinfo_t *info;

int lap_init(if_id)
int if_id;

char lap_name(if_id)
int if_id;

int lap_open(interface_name)
char *interface_name;

int lap_setinfo(if_id, info)
int if_id;
at_ifinfo_t *info;

int lap_shutdown(if_id)
int if_id;
```

DESCRIPTION

The ALAP library provides a generic interface that allows the user to control and examine AppleTalk interfaces. This is the preferred method of accessing and controlling ALAP interfaces (as opposed to `ioctl`(2) calls which are implementation and driver specific).

The library does not provide a way to read from and write to these interfaces; the `read`(2) and `write`(2) system calls should be used for this purpose. Any interface-dependent functions must also be accessed directly through implementation-dependent features of its interface driver.

All AppleTalk sockets (DDP naming space) are tied to a single interface, regardless of the number of physical interfaces installed. This interface is determined at system installation time, and the interface named is stored in the file `/etc/appletalkrc` (see `appletalkrc`(4) in *A/UX Programmer's Reference*).

The following ALAP library routines are available in `/usr/lib/libat.a`.

`lap_bind` registers an ALAP listener for the LAP packet type specified. The name specified must be a 1 to 13 character null-terminated string. Only one listener on a given LAP interface can be registered per type. In addition, only one ALAP type ID number can be used on a given ALAP interface; thus, preregistered ALAP type 1 (DDP short) and type 2 (DDP long) cannot be used by other processes other than through `ddp`(3N) library routines. Once a listener is registered, ALAP packets of the registered type can be read. The format of these packets (beyond the three bytes of the ALAP header) is specific to the interface type. See `ddp`(3N) for information on ALAP types 1 and 2.

The total number of ALAP types that can be bound to a given ALAP interface is limited. In the case of `localtalk0` (see below) this limit is 5, two of which are already in use.

You must be the superuser to use this routine.

if_id The file descriptor returned from a previous `lap_open` call.

type The ALAP packet type number. Currently known are 1 (DDP short) and 2 (DDP long). See *Inside AppleTalk* for details.

name A null-terminated string representing the registered *type*.

`lap_close` closes the interface file descriptor opened with `lap_open`. It only affects the local program.

if_id The file descriptor returned from a previous `lap_open` call.

`lap_default` returns a character pointer to the interface name of the default interface as defined in `/etc/appletalkrc`. It returns NULL on error.

`lap_name` returns a character pointer to the name of the specified interface. It returns NULL on error.

if_id The file descriptor returned from a previous `lap_open` call.

`lap_open` opens an AppleTalk networking interface and returns a file descriptor for a unique ALAP circuit.

interface_name
A null-terminated string which contains the interface name, such as `localtalk0`. Note that the interface name is actually a directory name located in `/dev/appletalk/lap` which contains the groups' special files.

`lap_getinfo` gets `at_ifinfo_t` ALAP and DDP information associated with the interface. DDP information is valid only if at least one interface is up and running.

if_id The file descriptor returned from a previous `lap_open` call.

info A pointer to `struct at_ifinfo` that will be filled with current ALAP and DDP information associated with this interface.

`lap_init` brings the AppleTalk specified interface online. You must be the superuser to use this routine.

if_id A file descriptor returned from a previous `lap_open` call.

`lap_setinfo` sets `at_ifinfo_t` information for the ALAP and DDP associated with the interface. All fields except `u.alap.node`, `u.alap.initial_node`, and `u.alap.rts_attempts` are ignored.

if_id A file descriptor returned from a previous `lap_open` call.

info A pointer to `struct at_ifinfo` that will be used to set ALAP and DDP characteristics associated with this interface.

`lap_shutdown` brings the specified AppleTalk interface offline. You must be the superuser to use this routine.

if_id A file descriptor returned from a previous `lap_open` call.

The ALAP and DDP information statistics is defined by the structure `at_ifinfo_t` defined in <at/lap.h>.

```
/* Link level info and statistics */

typedef struct at_ifinfo {
u_int   flags;          /* see AF_IFF below */
struct {
  /* Statistics for all interfaces types */
    u_long rcv_bytes;
    u_long rcv_packets;
    u_long xmit_bytes;
    u_long xmit_packets;
    u_long too_long_errors;
    u_long too_short_errors;
    u_long unknown_mblks;
    u_long ioc_unregistered;
    u_long type_unregistered;

/* Fields specific to individual interface types */
  union {
    struct {
       /* LocalTalk configuration */
        at_node node;
        at_node initial_node;
        u_int rts_attempts;

    /* LocalTalk statistics */
        u_long timeouts;
        u_long collisions;
        u_long crc_errors;
        u_long unknow_irupts;
        u_long overrun_errors;
        u_long abort_errors;
        u_long defers;
        u_long underrun_errors;
```

```
            u_long miss_sync_irupt;

        } alap;

    } lapinfo;

       struct {
           /* DDP configuration */
           u_short this_net;
           at_node this_node;
           at_node a_bridge;

           /* DDP statistics */
           /* receive errors */
           int socket_unregistered;
           int rcv_socket_outrange;
           int rcv_length_errors;
           int rcv_checksum_errors;
           /* transmit errors */
           int tag_room_errors;
       } ddpinfo;

    } at_ifinfo_t;

    /* Possible values for flags field of at_ifinifo_t */

    #define AT_IFF_IFMASK       0xfff
    #define AT_IFF_LOCALTALK    0x1

    #define AT_IFF_RUNNING      0x10000
    #define AT_IFF_DDP_RUNNING 0x20000
```

ERRORS

Unless otherwise noted, all routines return -1 on error with a detailed error code in `errno`.

`[ENOENT]`	No AppleTalk interface exists (`lap_default` and `lap_name`).
`[EINVAL]`	*if_id* is not a valid ALAP file descriptor (`lap_name`, `lap_bind`, `lap_setinfo`). Invalid listener name (`lap_bind`). Attempt to set invalid value (`lap_setinfo`).

[EEXIST]	*if_id* is not a valid ALAP file descriptor (lap_bind).
[EALREADY]	The interface is already online (lap_init). The interface is already offline (lap_shutdown).
[EACCESS]	The user must be the superuser (lap_init, lap_setinfo, and lap_shutdown).

See open(2), close(2), and ioctl(2) for additional errors.

FILES

```
/dev/appletalk/lap/*/...
/dev/appletalk/ddp/*
/etc/appletalkrc
```

SEE ALSO

close(2), ioctl(2), open(2), read(2), write(2), atp(3N), ddp(3N), nbp(3N), pap(3N), rtmp(3N), appletalkrc(4), appletalk(7); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

at_decompose_en, at_confirm_nve, at_deregister_name_nve, at_lookup_nve, at_nbp_shutdown, at_register_nve — AppleTalk® Name Binding Protocol (NBP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/nbp.h>

int at_decompose_en(entity, ent_len, object,
                    type, object_len,
                    type_len, zone, zone_len)
char *entity, *object, *type, *zone;
int ent_len, *object_len, *type_len, *zone_len;

int at_confirm_nve(object, object_len, type, type_len,
                   zone, zone_len, trys, secs, net,
                   node, socket)
char *object, *zone, *type;
int object_len, type_len, zone_len, trys, secs, net,
   node, socket;

int at_deregister_name_nve(object, object_len,
                           type, type_len)
char *object, *type;
int object_len;

int at_lookup_nve(object, object_len, type, zone,
                  type_len, zone_len, trys, secs)
char *object, *zone, *type;
int object_len, type_len, zone_len, trys, secs;

int at_nbp_shutdown()

int at_register_nve(object, object_len, type,
                    type_len, socket, trys, secs)
char *object, *type;
int object_len, type_len, socket, trys, secs;
```

DESCRIPTION

The NBP interface provides applications with access to the services of the AppleTalk Name Binding Protocol routines.

`at_decompose_en` decomposes an entity name string of the form

object:*type*@*zone*

into its *object*, *type*, and *zone* components, and returns their length.

entity A pointer to a character string containing the NVE name to be decomposed. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

ent_len The length of the entity-name character string. If this argument is zero, the length of the entity name will be calculated by treating it as a null-byte-terminated string.

object A pointer to a character string in which the object name will be returned. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

object_len The returned length of the object-name character string.

type A pointer to a character string in which the type name will be returned. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

type_len The returned length of the type-name character string.

zone A pointer to a character-string in which the zone name will be returned. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

zone_len The returned length of the zone-name character string.

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

`at_confirm_nve` is used to confirm that an entity is still registered at a given address. This is useful when time has elapsed between the lookup operation and the actual use of the AppleTalk internet address found in the lookup.

object A pointer to a character string that is the object name. This string is, in general, the one returned by the lookup operation to be confirmed.

object_len The length of the object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string. The string cannot be greater than AT_NBP_TUPLE_STRING_MAXLEN.

type A pointer to a character string that is the type name. This string is, in general, the one returned by the lookup operation to be confirmed.

type_len The length of the type-name character string. If this argument is zero, the length of the type-name string will be calculated by treating it as a null-byte-terminated string. The string cannot be greater than AT_NBP_TUPLE_STRING_MAXLEN.

zone A pointer to a character string that is the zone name. This string is, in general, the one returned by the lookup operation to be confirmed.

zone_len The length of the zone-name character string. If this argument is 0, the length of the zone-name string will be calculated by treating it as a null-byte-terminated string. The string cannot be greater than AT_NBP_TUPLE_STRING_MAXLEN.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If this argument is 0 (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing the repeat of the broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (60).

net The number of the network found in the lookup operation.

node The number of the node found in the lookup operation.

socket The number of the AppleTalk socket found in the lookup operation.

A value of -1 is returned when the lookup failed or when more than one NVE with a that name was found; 0 indicates that the NVE is no longer there; 2 indicates that there is an NVE with a

different *net* number or that there is an NVE with a different *socket* number; 3 indicates that there is an NVE with a different *node* number.

object A pointer to a string containing the object name.

object_len The length of this string.

type is a pointer to a string containing the object name.

type_len is the length of this string.

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned. An NVE is always deregistered if found. Errors range from an invalid NVE to an attempted deregistration of a valid NVE by a user other than the one that registered it.

`at_deregister_name_nve` is used to tell the NBP daemon to remove an NVE with the given *object* and *type* names from the list of NVEs for this node. Removing the NVE makes the resource represented by the NVE inaccessible to the network. You must have created the name (or be the superuser) in order to do this.

`at_lookup_nve` queries the NBP daemon for a list of NVEs that match the *object*, *type*, and *zone* name specification given in the arguments. It returns a count of the number of NVEs found that match the specification. A doubly-linked list of `typedef at_nve` is built and may be referred to via the external variables `at_nve_lkup_reply_head` and `at_nve_lkup_reply_tail`.

object A pointer to a character string that is the object name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

object_len The length of the object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string.

type A pointer to a character string that is the type name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

type_len The length of the type-name character string. If this argument is zero, the length of the type name string will be calculated by treating it as a null-byte-terminated string.

zone A pointer to a character string that is the zone name. The string can not be greater than AT_NBP_TUPLE_STRING_MAXLEN characters.

zone_len The length of the zone-name character string. If this argument is zero, the length of the zone-name string will be calculated by treating it as a null-byte-terminated string.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing a repeat of a broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (60).

Upon successful completion, the number of NVEs found is returned. If no NVEs were found, a value of 0 is returned. Upon error, a value of -1 is returned.

at_nbp_shutdown shuts down the NBP daemon. You must be the superuser to do this. Upon successful completion, a value of 0 is returned if the shutdown is successful. A value of -1 is returned if the user is not the the superuser or if there is a streams I/O error.

at_register_nve registers an NVE for the process with the NBP daemon. An NVE says, in effect, that this node on this AppleTalk socket has a resource of this type, named with this unique name.

object A pointer to a character string that is the object name. The string can not be greater than AT_NBP_TUPLE_STRING_MAXLEN characters.

object_len The length of the object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string.

type A pointer to a character string that is the type name. The string can not be greater than AT_NBP_TUPLE_STRING_MAXLEN characters.

type_len The length of a type-name character string. If this argument is zero, the length of the type-name string

will be calculated by treating it as a null-byte-terminated string.

socket The AppleTalk socket number on which to register this NVE.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing a repeat of the broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (60).

Upon successful completion, a value that is the NBP daemon registration number is returned. The registration number is used only to tell the NBP to deregister this NVE at a later time. Otherwise, a value of -1 is returned. There are two kinds of errors. If the global variable `at_nve_lkup_reply_count` is zero, the NVE did not respond. If `at_nve_lkup_reply_count` is nonzero, the NVE was not registered because the name is already in use. In that case, the duplicate name(s) are linked onto the list pointed to by `at_nve_lkup_reply_head` .

GLOBALS

`at_nve_lkup_reply_head`
This is a pointer to the head of the linked list of `at_nve` structures. f there are no NVEs found, this will have a value of NULL. If this variable has a non-NULL value upon entry to the `at_lookup_nve` function, it is assumed that it is pointing to a linked list of `at_nve` structures from a previous call to `at_lookup_nve` and the linked list members will be freed (via `free`(3)) first.

`at_nve_lkup_reply_tail`
This is a pointer to the tail of the linked list of `at_nve` structures. If there are no NVEs found, the pointer will have a value of NULL.

`at_nve_lkup_reply_count`
This is the count of the number of members in the linked list pointed to by `at_lookup_reply_head`.

ERRORS

All routines return -1 on error with a detailed error code in `errno`.

`[EBADF]`	*fd* is not a valid file descriptor.
`[ENOTTY]`	*fd* is not a TTY (that is, not a special device).
`[EINTR]`	The request was interrupted by signal.
`[ENXIO]`	Out of file descriptors.
`[EAGAIN]`	The request failed due to a temporary resource limitation; try again.
`[ENETDOWN]`	The network interface is down.
`[EPROTOTYPE]`	The protocol type requested is inappropriate for that AppleTalk socket
`[EBUSY]`	The requested service is unavailable at this time; try again.
`[EPERM]`	The user does not have permission to perform the requested task; user must be the superuser.
`[EINVAL]`	Designates an invalid argument.

See `open`(2), `close`(2), `read`(2), `write`(2), and `ioctl`(2) for additional error codes; see also errors returned in the underlying ATP, DDP and ALAP modules.

SEE ALSO

`close`(2), `ioctl`(2) `open`(2), `read`(2), `write`(2), `atp`(3N), `ddp`(3N), `lap`(3N), `pap`(3N), `rtmp`(3N), `appletalk`(7); *Inside AppleTalk;* "AppleTalk Programming Guide, in *A/UX Network Applications Programming*.

NAME

at_pap_close, at_papsl_deregister_nve, at_papsl_get_next_job, at_papsl_heres_status, at_papsl_init_nve, at_pap_open_nve, at_pap_read_ignore, at_pap_read, at_papsl_register_nve, at_papsl_status_nve, at_pap_write, at_pap_write_eof, at_pap_write_flush — AppleTalk® Printer Access Protocol (PAP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/pap.h>

void at_pap_close(socket)
int socket;

int at_papsl_deregister_nve(object, object_len,
                            type, type_len);
char *object, *type;
int object_len, type_len;

int at_papsl_get_next_job(fd)
int fd;

int at_papsl_heres_status(fd, status)
int fd;
char *status;

int at_papsl_init_nve(object, object_len, type,
                      type_len, trys, secs, status)
char *object, *type, *status;
int object_len, type_len, trys, secs;

int at_pap_open_nve(object, object_len, type,
                    type_len, zone, zone_len,
                    trys, secs, retry, name)
char *object, *zone, *name, *type;
int object_len, type_len, zone_len, trys, secs, retry;

int at_pap_read_ignore(fd)

int at_pap_read(fd, data, len)
int fd, len;
```

```
char *data;

int at_papsl_register_nve(fd, object, object_len,
                          type, type_len, trys, secs)
int fd, object_len, object_len, trys, secs;
char *object, *type;

int at_papsl_status_nve(object, object_len, type,
                        type_len, zone, zone_len,
                        trys, secs)
char *object, *type, *zone;
int object_len, type_len, zone_len, trys, secs;

int at_pap_write(fd, data, len)
int fd, len;
char *data;

int at_pap_write_eof(fd, data, len)
int fd, len;
char *data;

int at_pap_write_flush(fd, data, len)
int fd, len;
char *data;
```

DESCRIPTION

The PAP interface provides applications with access to the AppleTalk Printer Access Protocol operations.

`at_pap_close` closes an open PAP server or client AppleTalk socket.

socket The AppleTalk socket that is to be closed. It returns void upon completion.

`at_papsl_deregister_nve` deregisters a server's name previously registered by a call to `at_papsl_register_nve` or `at_papsl_init_nve`.

object A pointer to a string containing the object part of the name to be deregistered.

object_len The length of the object-name string. If *object_len* is zero, the string is assumed to be null-terminated and its true length is used.

type A pointer to a string containing the type part of the name to be deregistered.

type_len The length of the type-name string. If *type_len* is zero, the string is assumed to be null-terminated and its true length is used.

A value of 0 is returned if the name existed and was deleted. Otherwise, -1 is returned.

`at_papsl_get_next_job` is called by a server when it is ready to respond to a new PAP client. It returns a PAP-client AppleTalk-socket descriptor that is set up for PAP reading and writing from and to the client that has been waiting the longest.

fd A PAP-server AppleTalk socket descriptor from a previous `open`.

Upon successful completion, a PAP-client AppleTalk socket descriptor is returned. Otherwise, if the network has gone down since the last server access, -1 is returned

`at_papsl_heres_status` changes the `status` string associated with an open PAP-server AppleTalk socket. This is the string returned to a PAP-client from an `open` or `at_papsl_status_nve` call.

fd An open PAP-server AppleTalk socket returned from an `at_papsl_init_nve` call.

status A pointer to a null-terminated character string containing the `status` string being posted. Strings longer than 255 characters are truncated.

Upon successful completion, a value of 0 is returned. Otherwise, if the AppleTalk socket is no longer open, -1 is returned.

`at_papsl_init_nve` opens a PAP-server AppleTalk socket for a PAP server. At the same time, it registers a name for the server and posts a `status` string on it.

object A pointer to a string to be used as the object portion of the name being registered with NBP for this server. Wildcard names are not allowed.

object_len The length of the object. A length of zero means that the string is null-terminated and its true length should be used.

type A pointer to a string to be used as the type portion of the name being registered with NBP for this server.

Wildcard names are not allowed.

type_len The length of the type. A length of zero means that the string is null-terminated and its true length should be used.

trys The number of times the server name is searched for on the network before deciding the name is unique during registration.

secs The number of seconds between such attempts.

status A pointer to a null-terminated string used to post the server's status string.

Upon successful completion, the AppleTalk socket descriptor of the PAP server AppleTalk socket created is returned. Otherwise, -1 is returned.

`at_pap_open_nve` opens a PAP client AppleTalk socket to a server. It searches first for a server with a registered name that matches that described by the *object, type,* and *zone* parameters. The *object* and *type* may be wildcards (=). Though some PAP servers may be unavailable, `at_pap_open_nve` tries to access all available PAP servers the number of times specified by *retry* until one is found that will accept a client connection.

object A is pointer to a character string that is the object name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

object_len The length of an object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string.

type A pointer to a character string that is the type name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

type_len The length of the type-name character string. If this argument is zero, the length of the type-name string will be calculated by treating it as a null-byte-terminated string.

zone A pointer to a character string that is the zone name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

zone_len The length of a zone-name character string. If this argument is zero, the length of the zone-name string will be calculated by treating it as a null-byte-terminated string.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing a repeat of the broadcast request to look up this NVE. If this argument is zero (recommended), the NBP daemon will use a standard default value (60).

retry The number of times to search for a PAP server. During searching, a list of all available PAP servers is made, and a connection is attempted to each server in turn. This parameter specifies the number of times that all available PAP servers will be polled before the connection gives up. It does *not* specify the number of printers to be polled. If *retry* is less than zero, it retries indenfinitely.

name If this is non-NULL, the object name of the PAP server connected to it (maximum 32 characters) will be returned to the character array to which it points. If NULL, nothing is returned.

The global variable `at_pap_status` is a character array that contains the `status` string returned from the *last* connection request attempted. A special status string `No Printers` or `Unreachable` is returned when there are no printers registered, or the last printer registered did not respond, respectively.

Upon successful completion, this routine returns a PAP client AppleTalk socket connected to the server requested. Otherwise, -1 is returned.

`at_pap_read_ignore` issues a PAP read request and ignores any returned data. This is used to allow LaserWriters to function when they want to return "status" messages.

fd The streams file descriptor returned by `at_pap_open_nve`.

`at_pap_read` reads data from a client PAP socket opened by a `at_pap_open` or `at_papsl_get_next_job` call.

fd A PAP client AppleTalk socket descriptor from a previous open.

data The address of the data to be returned. The maximum data length returned is 512 bytes.

length The maximum length to be read.

Upon successful completion, the number of bytes read is returned; 0 is returned if an end-of-file has been reached; -1 is returned if the connection to the server has been broken.

`at_papsl_register_nve` registers a name for the PAP server described by the AppleTalk socket descriptor passed to it.

fd A PAP server AppleTalk socket descriptor for the server being registered.

object A pointer to a character string that is the object name. The string cannot be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

object_len The length of the object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string.

type A pointer to a character string that is the type name. The string cannot be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If the argument is zero (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing a repeat of the broadcast request to look up this NVE. If the argument is zero (recommend), the NBP daemon will use a standard default value (60).

A value of 0 is returned if the registration succeeded. A value of -1 is returned if either the AppleTalk socket is invalid or the name is already in use.

`at_papsl_status_nve` locates a PAP server and returns its `status` string. `at_pap_status` is a global array of characters that contains the returned device's `status` string.

object A pointer to a character string that is the object name. The string cannot be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

object_len The length of the object-name character string. If this argument is zero, the length of the object-name string will be calculated by treating it as a null-byte-terminated string.

type A pointer to a character string that is the type name. The string cannot be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

type_len The length of the type-name character string. If this argument is zero, the length of the type-name string will be calculated by treating it as a null-byte-terminated string.

zone A pointer to a character string that is the zone name. The string can not be greater than `AT_NBP_TUPLE_STRING_MAXLEN` characters.

zone_len The length of the zone-name character string. If this argument is zero, the length of the zone-name string will be calculated by treating it as a null-byte-terminated string.

trys The number of times that the NBP daemon will issue a broadcast request to look up this NVE. If the argument is zero (recommended), the NBP daemon will use a standard default value (5).

secs The number of seconds that the NBP daemon will wait before issuing a repeat of the broadcast request to look up this NVE. If the argument is zero (recommended), the NBP daemon will use a standard default value (60).

Upon successful completion, a value of 0 is returned; if no printer's status can be recovered, -1 is returned.

`at_pap_write` sends the data passed to it to the other end of a PAP client session.

fd A valid PAP client AppleTalk-socket descriptor from a call to `at_pap_open_nve` or

`at_papsl_get_next_job`.

data A pointer to the data being written.

len The length of the data being written; this must not exceed 512 bytes.

Upon successful completion, a value of 0 is returned; if the `write` cannot be completed, -1 is returned.

`at_pap_write_eof` sends the data passed to it to the other end of a PAP-client session. It also sends a PAP end-of-file indication to the other end to indicate that no more data will be sent. It does an implicit `at_pap_write_flush`.

fd A valid PAP client AppleTalk socket descriptor returned from a call to `at_papsl_get_next_job` or `at_pap_open_nve`.

data A pointer to the data being written.

len The length of the data being written; this must not exceed 512 bytes.

Upon successful completion, a value of 0 is returned; if the `write` cannot be completed, -1 is returned.

`at_pap_write_flush` sends the data passed to it to the other end of a PAP client session. Since PAP runs on top of ATP, PAP writes are queued up until either a complete ATP response is available (about 4 Kbytes) or an end-of-message is sent. This call sends an ATP end-of-message, which causes all waiting PAP writes to be sent to the other end. This should be done if a higher level protocol (for example, a handshake with a LaserWriter) needs to do a `write` followed by a `read`.

fd A valid PAP-client AppleTalk socket descriptor from a call to `at_papsl_get_next_job` or `at_pap_open_nve`.

data A pointer to the data being written.

len The length of the data being written; this must not exceed 512 bytes.

Upon successful completion, a value of 0 is returned; if the `write` cannot be completed, -1 is returned.

ERRORS

All routines return -1 on error with a detailed error code in `errno`.

`[EBADF]`	*fd* is not a valid file descriptor.
`[ENOTTY]`	*fd* is not a TTY (that is, not a special device).
`[EINTR]`	The request was interrupted by signal.
`[ENXIO]`	Out of file descriptors.
`[EAGAIN]`	The request failed due to a temporary resource limitation; try again.
`[ETIMEDOUT]`	The connection is timed out.
`[ESHUTDOWN]`	The requested AppleTalk socket has already been closed.
`[ENETDOWN]`	The network interface is down.

See `open`(2), `close`(2), `read`(2), `write`(2), and `ioctl`(2) for additional error codes; see also errors returned by the underlying NBP, ATP, DDP and ALAP modules.

SEE ALSO

`close`(2), `ioctl`(2), `open`(2), `read`(2), `write`(2), `atp`(3N), `ddp`(3N), `lap`(3N), `nbp`(3N), `rtmp`(3N), `appletalk`(7); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming*.

NAME

at_get_net_number, at_get_node_number, at_get_bridge_number — identify AppleTalk® node addresses

SYNOPSIS

```
#include <at/appletalk.h>

int at_get_net_number()
int at_get_node_number()
int at_get_bridge_number()
```

DESCRIPTION

The following routines allow the user to determine AppleTalk node addresses.

at_get_net_number returns the value of a node's current network number. A network number is supplied by an AppleTalk bridge node and is not built into the nodes on a network. Therefore, if there are no bridge nodes on the network, the network number will be zero.

Upon successful completion, a value from 0 through 65,535 (0xffff) is returned. Zero means that there is not an AppleTalk bridge node around to supply the network number. Network numbers are 16-bits (unsigned) and range from 1 through 65,535. Otherwise, -1 is returned.

at_get_node_number returns the node number of the node on which the current process is running. Node numbers are dynamically assigned by the ALAP layer when it is brought up. A node number lies in the range of 1 through 254.

Upon successful completion, a value from 1 through 254 is returned. Otherwise, -1 is returned.

at_get_bridge_number returns the number of the local bridge. A bridge number lies in the range of 1 through 254. If the value is 0, there is no bridge present.

SEE ALSO

atp(3N), ddp(3N), lap(3N), nbp(3N), pap(3N), appletalk(7); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

zip_getmyzone, zip_getzonelist — AppleTalk Zone Information Protocol (ZIP) interface

SYNOPSIS

```
#include <at/appletalk.h>
#include <at/atp.h>
#include <at/nbp.h>
#include <at/zip.h>

int zip_getmyzone(zone)
at_nvestr_t *zone;

int zip_getzonelist(start, buf)
int start;
at_nvestr_t *zone[];
```

DESCRIPTION

The ZIP interface provides applications with access to the AppleTalk Zone Information Protocol operations.

zip_getmyzone obtains the zone name for the local network. This routine sends a ZIP request to a local bridge for the zone name of the default network, and returns this zone name to the caller.

zone A pointer to the zone name. The zone string is defined by the following structure (see <at/nbp.h>):

```
typedef struct  at_nvestr {
        u_char  len;
        u_char  str[AT_NVE_STR_SIZE];
} at_nvestr_t;
```

len The size of the string in bytes.

str Contains the zone name.

This routine returns 0 upon success.

zip_getzonelist is used to obtain a complete list of all the zone names defined in the internet. This routine sends a ZIP request to a bridge for the list of zone names in the internet. The list is placed into the supplied buffer as concatenated at_nvestr_t structures.

start The starting index for the get zone list request. The start index is the value of the index at which to start, including zone names in the response. It is used to obtain a zone list that may not fit into one ATP The sesponse

packet. Th start index should initially be 1. While `zip_getzonelist` does not return 0, the caller must reissue `zip_getzonelist` calls, specifying a start index of the previous start index plus the previous return value of `zip_getzonelist`.

buf is a buffer to hold this list of zone names. Each zone name is an `at_nvestr_t` structure. The zone list buffer must be at least `AT_ATP_DATA_SIZE` bytes. Upon successful completion, this routines returns the number of zone names in the list. When all zones in the bridge's Zone Information Table have been returned, this routine returns 0.

DIAGNOSTICS

Both routines return -1 on error with a detailed error code stored in `errno`.

`[EINTR]`	The request was interrupted by signal (all).
`[EAGAIN]`	The request failed due to a temporary resource limitation; try again (all).
`[ENETUNREACH]`	A bridge node could not be found to process the request (all).
`[EINVAL]`	Invalid parameter (all).

Routines also return any error codes returned by the underlying ATP, DDP, or ALAP layers.

WARNINGS

The returned zone strings are not null-terminated.

SEE ALSO

ddp(3N), `lap`(3N), `nbp`(3N), `pap`(3N); `rtmp`(3N); *Inside AppleTalk;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

appletalkrc — AppleTalk® network configuration file

DESCRIPTION

The appletalkrc file contains information for configuring an AppleTalk network. The file is created at boot time by the AppleTalk startup routine. It does not need to be modified by a system administrator. The format of the file consists of a list of parameters and values, one per line:

parameter=value

Comments are indicated by a # character, and continue until the newline. The following parameters are defined.

interface Defines the interface which will host all DDP (Datagram Delivery Protocol) sockets on this system.

value A null-terminated string such as localtalk0.

No matter how many AppleTalk interfaces are active on the system, all DDP sockets are tied to one ALAP (AppleTalk Link Access Protocol) address space. Note that this is a DDP address-space and naming requirement; it has no relationship to routing of DDP packets through any particular ALAP interface.

EXAMPLE

The default appletalkrc file created by AppleTalk startup for a system with one AppleTalk card:

```
# AppleTalk configuration file
# Do not change the contents of this
# file while AppleTalk is active!

interface=  localtalk0   # DDP interface
```

FILES

/etc/appletalkrc
/etc/startup.d
/etc/newunix

SEE ALSO

appletalk(1M), newunix(1M), appletalk(7); "Installing and Administering AppleTalk," in *A/UX Network System Administration. Inside AppleTalk.* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*