# Open Transport AppleTalk Developer Note

PRELIMINARY
Revision 1.1b14
1/18/96

# Table of Contents

# Revision History

| | |
|---|---|
| 01/18/96 | Added DDP_OPT_SRCADDR option documentation |
| 01/12/96 | Added ServersStatus option to PAP |
| 11/30/95 | Update to reflect 1.1b10 |
| 07/18/94 | Update to reflect 1.0d13 |
| 06/28/94 | Partially revised to reflect 1.0d13 |
| 03/25/94 | Address formats revised |
| 03/15/94 | DDP Multi-Node addresses added |
| 03/10/94 | PAP added, option name correction |
| 02/14/94 | Updated |
| 10/19/93 | First compiled from the AppleTalk endpoint notes |

# Related Documents

*Inside AppleTalk®, Second Edition*, Gushuran S. Sidhu, et. at., Addison-Wesley Publishing, Inc.

*Apple Shared Library Manager Developer's Guide*, by ESD Publications, October 4, 1993, Apple Computer, Inc.

*Open Transport Client Developer Note*

# AppleTalk for Open Transport

This document describes how the Open Transport implementation of AppleTalk can be used by client applications. This document should be used along with the *Open Transport Client Developer Note*. That document describes general information about Open Transport endpoint libraries and Open Transport mapper libraries. However, it does not not provide any information specific to AppleTalk.

This document describes AppleTalk address formats, options specific to each kind of endpoint, and the Open Transport AppleTalk services library.

# Technical Specifications

This section describes AppleTalk address formats that are used in the Open Transport Endpoint functions (like *Snd(), Bind(),* etc.) Followed by the address formats are sections for each of the AppleTalk protocols that can be opened as endpoints and support the standard Open Transport endpoint functions. Finally, the AppleTalk services library is described. The AppleTalk services library is used for AppleTalk-related functions that are not transport independent and do not fit into the endpoint model.

The file OpenTptAppleTalk.h contains the declarations of the necessary constants and data structures needed. This file may be included by both 'C' and 'C++' source files. The MPW object files OpenTptATalk.o or OpenTptATalk.n.o must be linked with clients that are making AppleTalk specific Open Transport calls (but not AppleTalk-specific Option Management calls). The first file is for far-model MPW programs, and the second file is for near-model MPW clients.

# Address Formats

All of the AppleTalk endpoints handle both Name Binding Protocol (NBP) names and DDP addresses in places where the endpoint functions require the client to supply a protocol address.

## DDP Addresses

Open Transport AppleTalk allows four different forms of a DDP address. The primary form (8-bytes long) is called a DDPAddress and includes the DDP type. For all protocol layers above DDP, the client need not specify the DDP type when passing an address to an AppleTalk endpoint.

A DDPAddress looks like this:

| | |
|---|---|
| 2-byte | address type (AF_ATALK_DDP) |
| 2-byte | network number |
| 1-byte | node number |
| 1-byte | socket |
| 1-byte | DDP type |
| 1-byte | pad byte |

For C++ devlopers, the data structure has inline methods for getting and setting the various fields. These are: *Init(), SetNetwork(), SetNode(), SetSocket(), SetType(), GetNetwork(), GetNode(), GetSocket(),* and *GetType().*

The *Bind(), GetProtAddr(), ResolveAddr(),* and *RcvUData()* calls each return an address as the part of their return result. For each of these calls, the address format returned is a DDPAddress structure.

For example, a C++ code fragment that sends an echo request packet to a specific DDP address (network number $1234, node $87, socket $4, type $4) might look like this:

```
OSErr DoSend(TEndpoint* ep)
{
    DDPAddress  theDest;                            // Who to send to
    theDest.Init(0, 0xff, 4, 4);

    static unsigned char    theBuffer[] = "\x01Hello";  // What to send them

    TUnitData               unitData;                   // To pass to SndUData
    unitData.udata.Init(theBuffer, 6, 6);               // Initialize data part
    unitData.addr.Init(&theDest, sizeof(theDest));      // Initialize address part

    OSErr err = ep->SndUData(&unitData);
    if ( err != kOTNoError )
    {
        fprintf(stderr, "SndUData() returns %d\n", err);
    }
    return err;
}
```

## NBP name

The 2nd kind of address is an NBP address. It's structure is defined as the structure NBPAddress in OpenTptAppleTalk.h

An NBP address looks like this:

| | |
|---|---|
| 2-byte | addresstype(AF_ATALK_NBP) |
| n-bytes | NBP name string in form “name:type@zone” [NOT null-terminated] |

This form of an address can be used to specify the destination address in the following Open Transport calls: *SndUData(), SndURequest(), Connect(), Bind(), and ResolveAddr()*. The address is a character string of the form “<name>:<type>@<zone>“ where name, type, and zone are 32-characters maximum. The character “*” can be used for the zone name to specify the current zone. NBP wildcards, “=“ and “≈”, are not allowed in a name passed to any of the calls listed above. The string is neither a null-terminated “C” string nor a pascal string. The address is always referenced by a TNetbuf structure whose len field gives the length of the string.

If a client is going to use an NBP name repeatedly to send packets (DDP) or connectionless transactions (ATP), it is recommended that the client use the *ResolveAddr()* call first to resolve the NBP address into a DDP address and then use that address. Otherwise, for every packet or transaction, there could be an NBP lookup that occurs on the network.

A client can use the *ResolveAddr()* on any AppleTalk endpoint to resolve an NBP name into a DDP address. The address that is returned is a DDPAddress with the DDP type field set to zero.

If a client passes an NBP address to the *Bind()* call, that will cause the AppleTalk endpoint to open a dynamic DDP socket, and register the name on that socket. The full address that the name is registered on is returned in the address returned from *Bind()*.

## Combined Addresses

The 3rd address form is a combined address made up of both a DDP and NBP address. It's structure is defined as the structure DDPNBPAddress in OpenTptAppleTalk.h

A combined DDP / NBP address looks like this:

| | |
|---|---|
| 2-byte | address type(AF_ATALK_DDPNBP) |
| 2-byte | network number |
| 1-byte | node number |
| 1-byte | socket |
| 1-byte | DDP type |
| 1-byte | pad byte |
| n-bytes | NBP name string in form “name:type@zone” [NOT null-terminated] |

This form can be used anywhere the NBP address form can be used. An AppleTalk endpoint that is passed this form of an address by a client will check to see if the DDP portion of the address is valid. If so, it is used, otherwise, the NBP portion is used.

This form of an address can be used on the *Bind()* call if the client wants to bind the endpoint to a particular socket and register a name on that socket at the same time. More information about binding is supplied in the next section describing DDP.

As with an NBP name, a combined address is referenced by a TNetbuf whose len field gives the total length of the address.

## DDP Multi-Node Addresses

Open Transport AppleTalk supports a modified DDP address which may be used only by clients of DDP to Bind to multiple node addresses on the same physical port. This address takes the form of a normal DDPAddress (includes the space for a DDP type) however the first field is a "AF_ATALK_MNODE", signalling the DDP Multi-Node address. The only significant fields are the network number and the node number fields which the client may use to "suggest" an address to be used. The client need not specify either the socket or the DDP type since these fields are ignored for multi-node addresses. DDP will deliver any packet addressed to the bound multi-node [network:node] address, regardless of socket or DDP type. Multi-node clients must perform their own filtering if these two attributes are important. A future (post 1.0) version of DDP will likely support multinode in a more general way.

A DDP Multi-Node Address looks like this:

| | |
|---|---|
| 2-byte | address type(AF_ATALK_MNODE) |
| 2-byte | network number |
| 1-byte | node number |
| 1-byte | 0 |
| 1-byte | 0 |
| 1-byte | pad byte |

Multi-Node clients are internally copied on outgoing broadcast packets and self-send packets, reducing traffic on the net.

In version 1.1 of Open Transport, a multi-node endpoint must use the `DDP_OPT_SRCADDR` option to specify the source address for outgoing packets (see the section on DDP Options).

From C++, use the *Init(UInt16 network, UInt8 node)* function to initialize a DDP address to a multi-node address.

# Multihoming AppleTalk

The Open Transport implementation of AppleTalk has a significant feature not found in the classic AppleTalk implementation for the Macintosh. The Open Transport implementation supports multihoming (sometimes called multiporting.) It is possible for AppleTalk to be active on more than one network port on the machine at once.

The Network control panel selects which port is the “default” AppleTalk port. If an Open Transport client does not specify a port when creating an endpoint, the endpoint will be created on the default port. All calls through the classic AppleTalk device-driver based APIs will always use the default port.

There is no internal forwarding between ports when AppleTalk is active on more than one port. A client that opens an endpoint on one port will only be able to communicate with other nodes that can be reached through that port.

Some readers may know of two special cases of multihoming that the existing classic AppleTalk stack does support. The AppleTalk Internet Router supports multiple ports and routes traffic between them. And, Apple Remote Access supports two ports and handles internal forwarding so that data from an Appletalk client is sent out the right port.

NOTE: Multihoming is currently only available to Open Transport clients that speicifically request it by specifying the link layer when opening an AppleTalk endpoint.

# Using DDP

DDP is a connectionless datagram style of endpoint. It does not support any of the connection-oriented calls, nor any of the transaction calls.

A DDP endpoint can be created using *OTOpenEndpoint()* passing the DDP identification string constant *kDDPName*. For example:

```
OSErr err = OTOpenEndpoint(OTCreateConfiguration(kDDPName));
```

# Endpoint Information

DDP can send packets from 0 to 586 bytes long.

As with all connectionless datagram endpoints, expedited data is not supported for DDP.

By default, no checksum is performed on outgoing packets. However, any incoming packets containing a checksum are always checked. If the checksum does not match, the incoming packet will be discarded. If a client wants to put checksums on all outgoing packets, the OPT_CHECKSUM option can be specified in the options buffer of every TUnitData structure passed to the *SndUData()* call. Or, the client can use the *OptionMgmt()* call to turn on the checksum option, and DDP.will send checksums with all outgoing packets.

## Default Type

A new concept for the Open Transport implementation of AppleTalk is the *default* DDP type. Every DDP endpoint has a default type associated with it that can be specified at bind time. If a DDP type is specified when the client binds, that type becomes the default type. The default type has special significance for both sending and receiving packets.

When a client sends a packet and no type is specified in the address contained in the TSendUnitData structure passed to *SndUData()*, then the default type is used. If the DDP endpoint was not bound to a type, and the client does not specify one (or specifies zero as the type), DDP will not send the packet. If the client specifies a DDP type when sending a packet, that type overrides the default type

When receiving packets, if a type was specified at bind time ,then the client is guaranteed that all packets received will be of the same type. If no type was specified then all packet types addressed to the bound socket are accepted. After calling *RcvUData()* to receive an incoming packet, DDP will return to the client in the address field of the TUnitData, a DDPAddress structure containing the DDP address of the sender and the DDP type of that packet.

Multiple clients can bind with the same socket using different types and DDP will deliver incoming packets to the right client based upon the DDP type specified in the packet.

## Binding

The DDP bind operation associates an endpoint with an AppleTalk address. As with all connectionless endpoints, only one client may bind to any given address at the same time.

The basic rules of address binding are:

1) Static sockets are $01-$7F inclusive. Dynamic sockets are $80-$FE inclusive. Sockets $00 and $FF are invalid. Sockets $01-$3F are reserved for Apple's use only.

2) DDP node range is $01-$FE inclusive. On extended networks (Phase 2 networks), node $FE is reserved. Node $FF is used as the broadcast address. Node $00 is invalid.

3) If the client leaves the socket empty (or gives a zero-lenth address) when binding, DDP will allocate a dynamic socket (in the range $80 to $FF).

4) Only static sockets can be specified when binding. If a client tries to bind to a specific socket in the range $80 to $FF, the bind will fail with a kOTBadAddrErr.

5) Specifying a DDP type of $00 when binding will cause DDP to not filter packets based on type (i.e. all packets bound for the socket are delivered).

6) Specifying a DDP type when binding will cause DDP to set that type as the default for that endpoint. Only packets addressed for the bound socket with the default type will be delivered.

7) Binding to a socket using DDP type $00 ensures exclusive access to that socket (i.e. no other clients can bind with that socket).

8) Multiple clients can bind to the same static socket if a unique DDP type is used by each client.

9) Network and node numbers are ignored when binding.

Given the bind rules above, there are several ways a client can bind to an endpoint:

1) No socket and type are specified. (Either they are both zero in the requested address, or a zero-length address is requested.) This will cause DDP to allocate a dynamic socket for the client, set the default type to 0 and give the client exclusive access to the socket. All incoming packets bound for the allocated socket will be passed to the client. When sending, the client must specify the DDP type.

   AppleTalk endpoints above DDP in the protocol layers ignore the DDP type specified.

2) A static socket is specified but no type. The endpoint will be bound to the static socket and the endpoint has no default DDP type. All incoming packets bound for the socket will be passed to the client. When sending, the client must specify the DDP type.

3) No socket is given but a type is specified. DDP will assign a dynamic socket to the endpoint and set the endpoint's default DDP type to the type specified. Only incoming packets bound for the assigned socket with the default type will be passed to the client. When sending, the default DDP type will be used if the client does not specify a type.

   AppleTalk endpoints above DDP in the protocol layers ignore the DDP type, so this case is the same as #1 for all AppleTalk endpoints except DDP.

4) A static socket and a type are specified. The endpoint will be bound to the socket and type specified. The type specified will become the endpoint's default DDP type. Only incoming packets addressed to the specified socket and type will be delivered to the client. When sending, the default DDP type will be used if the client does not specify a type.

   AppleTalk endpoints above DDP in the protocol layers ignore the DDP type, so this case is the same as #2 for all AppleTalk endpoints except DDP.

5) Multiple binds to a static socket using unique types. This is an extension of case 4 above with DDP de-multiplexing the packets.

   AppleTalk endpoints above DDP in the protocol layers ignore the DDP type, so this case is valid only for DDP endpoints.

6) An NBP address is passed. DDP will assign a dynamic socket and register the name on that socket. The endpoint will have no default DDP type. If the name already exists on the network, the bind will fail.

7) A combined NBP and DDP address is passed. DDP treats this in two parts. First, the DDP tries to bind to the DDP address according to the first five of these scenarios. If the bind is successful, then the NBP name is registered on the socket that the endpoint was bound to.

See the *Open Transport Client Developer Note* , for details on the specific calls to *Bind().*

# Options

This section documents the options that DDP supports in the current release of Open Transport.

## The OPT_CHECKSUM Option

The OPT_CHECKSUM can be used two ways. The client can specify this option on every call to *SndUData()* and control the sending of DDP checksums on a per packet basis, or the client can use the *OptionMgmt()* function to enable or disable checksums for all outgoing packets.

By default, a checksum is not performed.

## The DDP_OPT_SRCADDR Option

The DDP_OPT_SRCADDR is used to override the source address on an outgoing packet. This option is only allowed on a per-packet basis, and may not be used in the *OptionMgmt()* function call. The option must be a DDP Address structure using the AF_ATALK_DDP address format. The source network number, node number, and source socket will be taken from the DDP Address structure. It is an error for these values to be illegal.

This option is most often used in conjuction with a multi-node endpoint, but it may also be used on normal endpoints.

# Using ATP

ATP is a connectionless transaction endpoint. It does not support any of the connection-oriented endpoint calls.

When sending a transaction request, ATP sends an Exactly Once (XO) request if the T_ACKNOWLEDGED bit is set for the transaction, and sends an At Least Once (ALO) request if it is not. ATP treats the first four bytes of data that the client specifies as the ATP User Bytes and places them in the ATP header of the outgoing request. If the client does not specify at least four bytes of data in the request, the user bytes are padded with zeros. Similarly, the 4 bytes in the user bytes portion of the ATP header of the first reply packet are placed in the first four bytes of the user's response buffer. The 4 bytes in the user bytes portion of the ATP header for responses after the first response packet are ignored. Note that if one endpoint sends a request containing less than 4 bytes of data, the responding side will receive a request containing 4 bytes of data. The first 4 data bytes of a client's reply data will be used as the user bytes for each ATP reply packet.

ATP imposes no limit on the number of outstanding transactions that a client may generate. A client may have several ATP transactions

Options are available to configure ATP to to set retry count and interval between retries, as well as the release timer setting. These are described below.

The maximum packet data length for ATP is dependent on the layer running underneath it. In the AppleTalk environment, with DDP as the datagram delivery protocol, ATP may have up to 578 data bytes in any given packet. This does not include the 4 ATP header bytes or the 4 bytes of ATP User data which precede the data portion of all packets. When running over DDP, ATP request packets may contain the 582 data bytes, and ATP reply buffers may contain up to 4628 data bytes.

When sending a request, the client may specify any of the AppleTalk address types discussed earlier (DDP, NPB, combined)

# Options

ATP options are provided to enable the client to configure ATP behavior in several areas.

ATP supports the generic option `OPT_RETRYCNT` to set the number of times a request will be retried before giving up and returning an error to the client. (Default setting is **8 retries**.)

ATP also supports the option `OPT_INTERVAL` which enables the client to specify how long an interval should exist between retry attempts. (Default setting is **2 seconds**.)

ATP also supports the option `ATP_OPT_RELTIMER` which enables the client to specify how long to wait for an **ATPRelease** before discarding outgoing response data packets. The acceptable values for this field are {**0**=30 secs; **1**=1 minute; **2**=2 minutes; **3**=4 minutes; **4**=8 minutes}. (Default setting is **30** seconds.)

ATP also supports the option `ATP_OPT_REPLYCNT`, which specifies the number of replies (1 through 8) expected to a request. Eight (8) is the default. If the responder is not going to set the EOM bit in the message, it is vital that this option be used so that ATP knows not to wait for more responses.

# Using ADSP

ADSP is a connection-oriented stream. It does not support any of the connectionless datagram or transaction endpoint calls.

When initiating a connection, the client may specify any of the AppleTalk address types discussed earlier (DDP, NPB, combined).

ADSP supports two data channels. Normal data and expedited data. Not all connection-oriented streams support an expedited data channel so clients that wish to run over other stream protocols should not use ADSP's expedited (or "attention") channel.

By default, ADSP's normal data channel does not support the T_MORE flag. ADSP send and receives a continuous stream of bytes with no message delimiters. This is discourage the use of ADSP's end-of-message facility since not all stream protocols support this facility. In this default case, all calls to *Rcv()* will return with the T_MORE bit set in the flags, and the T_MORE bit is ignored on all calls to *Snd()*. Through the use of option management, the client can use the OPT_ENABLEEOM option to turn on ADSP's EOM facility.

After ADSP's EOM facility has been turned on, ADSP supports an infinite length Transport Data Service Units (TSDU) on the normal data channel. It also support zero-length TSDUs. This means that a client can send any number of bytes before calling *Snd()* without passing the T_MORE flag. The client may also call *Snd()* with no data and without the T_MORE flag set. This will be carried across the transport to the remote client; that client will receive a zero-length read without the T_MORE flag set.

The expedited channel supports a TSDU size of either 570 or 572 (See below). And, depending upon how the options are set up, it also supports zero-length TSDU's.

The expedited channel is a little unusual since the ADSP protocol defines an "attention code", a two-byte value from $0000 to $EFFF, that goes along with each unit of expedited data. The current implementation is to treat these two bytes as the first part of each expedited TSDU. This has the following properties / problems:

- The minimum TSDU is two bytes. If the client sends zero or one byte, then the data is padded out to two bytes before being transmitted. The client on the receiving side may not be expecting this behavior if it is a transport independent application and isn't coded specifically for ADSP.
- The client is responsible for ensuring the first two bytes are not in the reserved range $F000 to $FFFF.

Other alternatives for handling the attention channel are being explored. Some of the alternatives include:

- Ignore the attention code altogether. This prevents the client from talking to another ADSP client where the attention codes are meaningful. (This implies the application is not transport independent.)
- Use option management to set the attention code before every send on the attention channel, if control over the attention code is desired. The client would use option management to read the attention code on incoming expedited data.
- Use an option that changes the behavior of ADSP from ignoring the attention code altogether for clients that are transport independent, to treating the attention code as the

first two bytes of the TSDU. This is most likely the way the ADSP endpoint will treat the attention channel.

# Options

ADSP supports two options:

## The OPT_CHECKSUM Option

By default, outgoing ADSP packets do not have DDP checksums. The client can force ADSP to send all outgoing packets with DDP checksums by using the *OptionMgmt()* call to turn on the OPT_CHECKSUM option.

## The OPT_ENABLEEOM Option

A client can enable ADSP's end-of-message facility by using the *OptionMgmt()* call to turn on the OPT_ENABLEEOM option. This option is remembered internally on a per endpoint basis. One ADSP endpoint may have EOM enabled, and an another endpont may not.

# Using PAP

PAP is a connection-oriented stream. It does not support any of the connectionless datagram or transaction endpoint calls.

When initiating a connection, the client may specify any of the AppleTalk address types discussed earlier (DDP, NPB, combined).

PAP supports a single data channel and has OrderlyRelease capability.

By default, PAP's data channel does not support the T_MORE flag. PAP sends and receives a continuous stream of bytes with no message delimiters. This is to discourage the use of PAP's end-of-message facility since not all stream protocols support this facility. In this default case, all calls to *Rcv()* will return with the T_MORE bit set in the flags, and the T_MORE bit is ignored on all calls to *Snd()*. Through the use of option management, the client can use the OPT_ENABLEEOM option to turn on PAP's EOM facility.

After PAP's EOM facility has been turned on, PAP supports an infinite length Transport Data Service Units (TSDU) on its data channel. It also support zero-length TSDUs. This means that a client can send any number of bytes before calling *Snd()* without passing the T_MORE flag. The client may also call *Snd()* with no data and without the T_MORE flag set. This will be carried across the transport to the remote client; that client will receive a zero-length read without the T_MORE flag set.

Due to its evolution, PAP exhibits somewhat different behavior, depending on whether it is acting as a server or a workstation. In its workstation form, PAP requests connections, sends data, and closes connections pretty much as expected. When performing a server function, PAP has some additional responsibility in the world of connection arbitration. A PAP server, upon receiving an OpenConnection request must delay granting the request for some fairly arbitrary period of time (nominally 2 seconds), accumulating any additional requests which come in and then, at the end of the waiting period, grant the request of the workstation which claims to have been waiting the longest, via a "wait time" field in the PAP header of the packet. On the workstation side of this equation, in order to support this requirement, PAP tracks the elapsed time for each endpoint which has been opened and fills out the "wait time" field appropriately so that if a client has to try multiple times to connect to a busy LaserWriter, the endpoint remembers the accumulated time since the first attempt and reports that to the LaserWriter on every subsequent connection attempt.

# Options

PAP supports two options:

## The PAP_OPT_OPENRETRY Option

By default, outgoing PAP OpenConnection packets will not be retried by PAP. (ATP will retry the request per its configuration options.) The client can force PAP to retry outgoing OpenConnection packets by using the *OptionMgmt()* call to specify a retry count in the PAP_OPT_OPENRETRY option.

## The OPT_ENABLEEOM Option

A client can enable PAP's end-of-message facility by using the *OptionMgmt()* call to turn on the OPT_ENABLEEOM option. This option is remembered internally on a per endpoint basis. One PAP endpoint may have EOM enabled, and an another endpont may not.

## The OPT_SEVERSTATUS Option

This option is used to set the status string returned by the server in response to a SendStatus request from a client. This option is remembered internally on a per socket basis. The value of this option is the string to return. The lenght of the string must be in the range 0 – 255 bytes, and it is not preceeded by a lenght byte.

# The AppleTalk Services Library

A client of the OpenTransport implementation of AppleTalk may need to access some of AppleTalk's features that are not available through the standard Open Transport Endpoint Library. For example, the standard endpoint API's do not provide a way for a client to find out any information relating to AppleTalk zones, network numbers, or router addresses.

In order to provide this information, there is an Open Transport API specific to AppleTalk called the AppleTalk services library. Like other Open Transport libraries such as the mapper and endpoint libraries, the AppleTalk services library supports the basic Open Transport Library provider functions:

InstallNotifier()
GetNotifier()
RemoveNotifier()
SetSynchronous()
SetAsynchronous()
IsSynchronous()

These functions are described in the Open Transport Client Developer's Note.

In addition to these basic functions, the AppleTalk services library has the following functions:

GetATalkInfo()
GetMyZone()
GetLocalZones()
GetZoneList()

The functions *OTOenAppleTalkServices()* and *OTCloseProvideer()* create and destroy an AppleTalk services object. More than one client can create an AppleTalk services object, and a client can create more than one if desired. If the AppleTalk services object is used in asynchronous mode (recommmended), it allows only one call of each type to be outstanding at any time on any single AppleTalk services object. In order to use the object in asynchronous mode, the client must use *InstallNotifier()* to install a notifier routine that will be called when a call completes.

The AppleTalk Services calls are described on the following pages.

# OTOpenAppleTalkServices

**FUNCTION**

OTOpenAppleTalkServices Create an AppleTalk services object.

**C INTERFACE**

```
pascal ATSvcRef OTOpenAppleTalkServices(OTConfiguration* path,
            OTOpenFlags flags, OSErr* err);
```

**C++ INTERFACE**

none

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| path | x | / |
| flags | x | / |
| err | x | (x) |

`OTOpenAppleTalkServices` creates an AppleTalkServices object and returns a reference to the client.

The client may pass a copy (see OTCloneConfiguration in the Open Transport Client Note) of the OTConfiguration* that was used to open an AppleTalk endpoint to create an AppleTalkServices object using the same hardware port as the endpoint. kDefaultAppleTalkServicesPath may also be passed, in which case, the object will be created on the default hardware port (the one chosen using the Network CDev). The flags parameter is currently ignored and should be zero. The err parameter should point to a variable of type OSErr. If the call completes succesfully, a reference to the AppleTalk services object that was just created is returned and the error variable will be set to zero. If an error occurs, the reference returned will be zero, and the error variable will be set.

**RESULT CODES**

%%%%

**SEE ALSO**

`OTAsyncOpenAppleTalkServices`

# OTAsyncOpenAppleTalkServices

**FUNCTION**

`OTAsyncOpenAppleTalkServices` Create an AppleTalk services object asynchronously.

**C INTERFACE**

```
OSErr       OTAsyncOpenAppleTalkServices(OTConfiguration* config,
            OTOpenFlags oflag, OTNotifyProcPtr proc, void* contextPtr)
```

**C++ INTERFACE**

None. (C++ clients should use the C interface to this function.)

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| config | x | / |
| oflag | x | / |
| proc | x | / |
| contextPtr | x | / |

`OTAsyncOpenAppleTalkServices` creates an AppleTalk services object asynchronously, based on the supplied information. If this function returns an error immediately, then the noticiation function will not be called. If `kOTNoError` is returned, then the notification function will be called with the results of the open.

The `config` and `oflag` parameters have the same meaning as for `OpenAppleTalkServices`.

When the open is complete, your notification function will be called with the `code` parameter set to `T_OPENCOMPLETE`. The `result` parameter will either be `kOTNoError` if the open was successful, or will return a result code describing the error. If the open was successful, the cookie is the `EndpointRef` for the endpoint that was opened.

**Warning**: The OTAsync`OpenAppleTalkServices` function destroys the OTConfiguration returned by OTCreateConfiguration. Never attempt to use the same configuration to open multiple endpoints. You can use the `OTCloneConfiguration` function to clone the configuration for this purpose.

**RESULT CODES**

`kOTBadFlagErr`

`kOTBadNameErr`

`kOTCancelledErr`

**SEE ALSO**

`OpenAppleTalkServices`

# GetMyZone

**FUNCTION**

GetMyZone Get the current zone

**C INTERFACE**

```
pascal OSErr OTATalkGetMyZone(ATSvcRef ref, TNetbuf* zone);
```

**C++ INTERFACE**

```
pascal OSErr TAppleTalkServices::GetMyZone(TNetbuf* zone);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| zone->maxlen | x | / |
| zone->len | / | x |
| zone->buf | x | (x) |

GetMyZone is used to determine this machine's current AppleTalk zone. The current zone will be put into the TNetbuf pointed to by the zone parameter.

If the AppleTalk services object is in asynchronous mode, the client's notifier will be called with a T_GETMYZONECOMPLETE event and the cookie parameter to the notifier will contain the zone parameter.

The zone returned in the TNetbuf will be a pascal-style string of at most 32 characters. Using a pascal-style string is redundant since the length of the string could be determined from the maxlen field of the TNetbuf, but the other zone-related calls use pascal-style strings, so this call also uses them for consistency.

**RESULT CODES**

kOTBadReferenceErr

kOTNoDataErr

kOTBufferOverflowErr

**SEE ALSO**

GetZoneList, GetLocalZones

# GetZoneList

**FUNCTION**

GetZoneList — Get the list of zones available on the AppleTalk network.

**C INTERFACE**

```
pascal OSErr OTATalkGetZoneList(ATSvcRef ref, TNetbuf* zones);
```

**C++ INTERFACE**

```
pascal OSErr TAppleTalkServices::GetZoneList(TNetbuf* zones);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| zones->maxlen | x | / |
| zones->len | / | x |
| zones->buf | x | (x) |

GetZoneList is used to determine the current list of AppleTalk zones The zone list will be put into the TNetbuf pointed to by the zones parameter.

If the AppleTalk services object is in asynchronous mode, the client's notifier will be called with a T_GETZONELISTCOMPLETE event and the cookie parameter to the notifier will contain the zones parameter.

The zones returned in the TNetbuf will be a series of pascal strings placed one after the other.

**RESULT CODES**

kOTBadReferenceErr

kOTNoDataErr

kOTBufferOverflowErr

**SEE ALSO**

GetMyZone, GetLocalZones

# GetLocalZones

**FUNCTION**

GetZoneList — Get the list of zones available on the local network cable.

**C INTERFACE**

```
pascal OSErr OTATalkGetLocalZones(ATSvcRef ref, TNetbuf* zones);
```

**C++ INTERFACE**

```
pascal OSErr TAppleTalkServices::GetLocalZones(TNetbuf* zones);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| zones->maxlen | x | / |
| zones->len | / | x |
| zones->buf | x | (x) |

GetLocalZones is used to determine the list of AppleTalk zones available on the current network cable. The zones will be put into the TNetbuf pointed to by the zones parameter. For a non-extended network (e.g. LocalTalk), this call will return only one zone, the same one returned by GetMyZone.

If the AppleTalk services object is in asynchronous mode, the client's notifier will be called with a T_GETLOCALZONESCOMPLETE event and the cookie parameter to the notifier will contain the zones parameter.

The zones returned in the TNetbuf will be a series of pascal strings placed one after another

**RESULT CODES**

kOTBadReferenceErr

kOTNoDataErr

kOTBufferOverflowErr

**SEE ALSO**

GetMyZone, GetZoneList

# GetInfo

**FUNCTION**

ATalkGetInfo Get information about AppleTalk.

**C INTERFACE**

```
pascal OSErr OTATalkGetInfo(ATSvcRef ref, TNetbuf* info);
```

**C++ INTERFACE**

```
pascal OSErr TAppleTalkServices::GetInfo(TNetbuf* zones);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| info->maxlen | x | / |
| info->len | / | x |
| info->buf | x | (x) |

GetInfo is used to query AppleTalk for information about the current environment. This call returns the machine's DDP address, the address of a local router, the current cable range for the cable the machine is connected to, and whether or not the current network link is extended (Phase 2) or not .

If the AppleTalk services object is in asynchronous mode, the client's notifier will be called with a T_GETATALKINFOCOMPLETE event and the cookie parameter to the notifier will contain the info parameter.

The TNetbuf will be filled in with a structure that looks like:

```
struct AppleTalkInfo
{
        DDPAddress      fOurAddress;
        DDPAddress      fRouterAddress;
        UInt16                  fCableRange[2];
        UInt16                  fFlags;
};
//
// fFlags is bitmapped:
//
enum
{
        kATalkInfoIsExtended            = 0x0001,
        kATalkInfoHasRouter             = 0x0002
};
```

The fOurAddress contains the machines DDP address.

If the kATalkInfoHasRouter bit is set in the fFlags field, then the DDP address of a router on the same cable as this machine is contained in the fRouterAddress field.

The cable range (or network range) for the cable this machine is connected to is returned in the fCableRange array.

If the current network link is an extended network (AppleTalk Phase 2), then the kATalkInfoIsExtended bit in the fFlags field will be set.

**RESULT CODES**

```
kOTBadReferenceErr
kOTBufferOverflowErr
```

**SEE ALSO**

```
none
```

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

# The Mapper library and AppleTalk

This section is an addendum to the information in the *Open Tranport Client Developer Note* which describes the Open Transport mapper library. This section describes how a mapper behaves for AppleTalk and the formats of data passed to and returned from the calls.

When using the *OTOpenMapper()* to create a mapper, use the constant kNBPName when creating an OTConfiguration to pass as the first parameter to *OTOpenMapper()*.

A Mapper created on an AppleTalk network supports all three mapper calls: *RegisterName(), DeleteName(), andLookUpName(),*.

## Name formats

As an input parameter to all four calls, an NBP name format is as follows:

*object name*:**:***object type***@***zone name*

For example:

FreddyPrinter**:**LaserWriter**@**Lake Wobegon

and

Ronnies Mac Plus**:**Workstation**@***

The restrictions are that each component of the name may be up to 32 characters long for a total length of 98 characters including the separators (":" between the object name and object type; and "**@**" between the object type and the zone name.) When the name is stored in a netbuf the netbuf *len* field refers to the total length of the name string including separators.

Open Transport defines the "\" character to be an escape charatcer for NBP names. This allows the special characters ":", "@" and "\" to be used anywhere in an NBP name without confusion by preceeding the special character with a "\" character. For example: "Test\:1:Workstation\@Foo@Zone\\RD1" corresponds to an NBP entity with the name "Test:1", type "Workstation@Foo", and zone "Zone\RD1".

For *LookUpName()*, there may be more than one name returned in the output TNetbuf. The names are returned as a group of structures that each look like:

```
short   length of address field (= 8)
short   length of name field
short   address type(AF_ATALK_DDP)
short   network number
byte    node
byte    socket
byte    type (always zero)
byte    pad byte
bytes   “<name>:<type>@<zone>“
pad bytes to quad boundary
```

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

The structures are padded so that each one begins on a multiple of four bytes away from the start of the previous one. For example, given a pointer to the start of one name, the start of the next name is computed:

```
Ptr = Ptr + ((length + 3) & ~3);
```

The C or C++ functions / methods for an NBPEntity or DDPNBPAddress can be used to parse the name, if desired.

## Wild Cards

AppleTalk names may be "wild carded" when used in the *LookUpName()* call.

The entire object name and / or object type fields may be wild carded by the "=" character to match anything

i.e. Freddy:= matches any object named Freddy in this zone

=:Workstation@* matches any Workstation in this zone

The object name and / or object type fields may be partially wild carded using at most one "≈" character anywher in the name and / or type field. A single "≈" character in a field is equivalent to "=".

i.e. A≈:=@* matches any object with a name beginning with "A" in this zone

=:Laser≈ matches any object with an object type zone beginning with Laser *[would include LaserWriters, LaserShare spoolers, Laserprinters, etc.]*

Wild cards may not be used for the *RegisterName() or DeleteName(),* calls.

## Zone names

The current zone is frequently represented by the "*" character which is a form of wild carding. This is entirely optional. A zone of "*" and no zone are considered by NBP to be the same thing.

## NBPEntity

The NBPEntity structure allows a convenient way to manipulate an AppleTalk NBP name. It's use is not required to manipulate NBP addresses under Open Transport, but it is provided as a convenience for porting programs written for classic AppleTalk. The definition of this structure is:

```
struct NBPEntity
{
        UInt8   fEntity[kNBPMaxEntityLength];
};
```

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTInitDDPAddress

**FUNCTION**

OTInitDDPAddress Initialize an NBPAddress structure

**C INTERFACE**

```
pascal void OTInitDDPAddress(DDPAddress* address, UInt16 net, UInt8
          node, UInt8 socket, UInt8 ddpType);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| address | x | (x) |
| net | x | / |
| node | x | / |
| socket | x | / |
| ddpType | x | / |

`OTInitDDPAddress` can be used to initialize a `DDPAddress` structure with the net, node, socket and ddpType information provided in the call.

**SEE ALSO**

`OTInitNBPAddress, OTInitDDPNBPAddress, OTCompareDDPAddresses`

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTInitNBPAddress

**FUNCTION**

OTInitNBPAddress Initialize an NBPAddress structure

**C INTERFACE**

```
pascal void OTInitNBPAddress(NBPAddress* address, const char* name);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| address | x | (x) |
| name | (x) | / |

`OTInitNBPAddress` can be used to initialize an NBP address with the name specified in `name`. The `name` parameter is assumed to already be in the canonical format described in the previous section. The function returns the size of the `NBPAddress`, which is basically four (the size of the `OTAddressType` field) plus the length of the string at `name`.

**SEE ALSO**

`OTInitDDPAddress, OTInitDDPNBPAddress`

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTInitDDPNBPAddress

**FUNCTION**

OTInitDDPNBPAddress Initialize a DDPNBPAddress structure

**C INTERFACE**

```
pascal size_t OTInitDDPNBPAddress(DDPAddress* address, const char* name,
            UInt16 net, UInt8 node, UInt8 socket, UInt8 ddpType);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| address | x | (x) |
| name | (x) | / |
| net | x | / |
| node | x | / |
| socket | x | / |
| ddpType | x | / |

`OTInitDDPNBPAddress` can be used to initialize a `DDPNBPAddress` structure with the name, net, node, socket and ddpType information provided in the call. It returns the resulting size of the address structure, which is just the length of the `name` parameter, plus the size of a `DDPAddress`.

**SEE ALSO**

`OTInitNBPAddress, OTInitDDPAddress`

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTCompareDDPAddresses

**FUNCTION**

OTCompareDDPAddresses Compare two DDPAddresses

**C INTERFACE**

```
pascal Boolean OTCompareDDPAddresses(const DDPAddress* addr1, const
            DDPAddress* addr2);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| addr1 | (x) | / |
| addr2 | (x) | / |

OTCompareDDPAddresses compares two DDP addresses for equality. It will only compare DDP addresses (not NBP or DDPNBP addresses). Other address types will always return false. It uses the zero-matches-anything rule that is part of the AppleTalk specification when doing the matching. The function returns true if the two addresses match.

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTInitNBPEntity

**FUNCTION**

OTInitNBPEntity Initializes an NBPEntity structure

**C INTERFACE**

```
pascal void OTInitNBPEntity(NBPEntity* entity);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | x | (x) |

OTInitNBPEntity initializes an NBPEntity structure. It sets the NBP name, type and zone to empty strings.

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTGetNBPEntityLengthAsAddress

**FUNCTION**

OTGetNBPEntityLengthAsAddress Returns the number of bytes needed to store an NBPEntity into an NBPAddress or DDPNBPAddress.

**C INTERFACE**

```
pascal size_t OTGetNBPEntityLengthAsAddress(const NBPEntity* entity);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | / |

`OTGetNBPEntityLengthAsAddress` returns the number of bytes required to store an `NBPEntity` into an `NBPAddress` or `DDPNBPAddress`. This allows the proper sizing of the buffer.

**SEE ALSO**

`OTSetAddressFromNBPEntity`

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetAddressFromNBPEntity

**FUNCTION**

OTSetAddressFromNBPEntity Stores an NBPEntity as an NBP address string.

**C INTERFACE**

```
pascal size_t OTSetAddressFromNBPEntity(UInt8* namebuf, const NBPEntity*
            entity);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| namebuf | x | (x) |
| entity | (x) | / |

`OTSetAddressFromNBPEntity` stores the information in the `NBPEntity` into the buffer namebuf in the format required for the Mapper calls. It returns the number of bytes that were used in the buffer (Use OTGetNBPEntityLengthAsAddress to determine the number of bytes needed ahead of time). This function will handle all of the "escaping" needed for the name format

**SEE ALSO**

`OTGetNBPEntityLengthAsAddress`, OTSetNBPEntityFromAddress

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetAddressFromNBPString

**FUNCTION**

OTSetAddressFromNBPString Copies an NBP address string into a buffer

**C INTERFACE**

```
pascal size_t OTSetAddressFromNBPString(UInt8* namebuf, const char*
            nbpName, SInt32 len);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| namebuf | x | (x) |
| nbpname | (x) | / |
| len | x | / |

`OTSetAddressFromNBPString` will copy the string `nbpName` into the buffer namebuf. The `len` parameter indicates the number of characters to copy. If the `len` parameter is -1, then the length of the `nbpName` string will be used for the copy. The number of bytes actually copied is returned.

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetNBPEntityFromAddress

**FUNCTION**

OTSetNBPEntityFromAddress Parses and stores the name in an NBPAddress or DDPNBPAddress into an NBPEntity.

**C INTERFACE**

```
pascal size_t OTSetNBPEntityFromAddress(NBPEntity* entity, const UInt8*
            addrBuf, size_t len);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | x | (x) |
| addrBuf | (x) | / |
| len | x | / |

OTSetNBPEntityFromAddress parses an NBPAddress or DDPNBPAddress into the NBP name, type and zone, and stores the result into an NBPEntity. From the NBPEntity, each of the consituent parts of the name may be easily retrieved or changed.

**SEE ALSO**

`OTGetNBPEntityLengthAsAddress`, OTSetAddressFromNBPEntity, OTSetNBPName, OTSetNBPType, OTSetNBPZone, OTExtractNBPName, OTExtractNBPType, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetNBPName

**FUNCTION**

OTSetNBPName Set the NBP name portion in an NBPEntity structure.

**C INTERFACE**

```
pascal Boolean OTSetNBPName(NBPEntity* entity, const char* name);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | (x) |
| name | (x) | / |

`OTSetNBPName` will store the NBP name specified by the `name` parameter into the `NBPEntity entity`, deleting any previous name stored there. The name supplied should NOT have any of the NBP escape charaters stored in it. This function returns false if the name parameter is longer than the maximum allowed NBP name (32 characters).

**SEE ALSO**

OTSetNBPType, OTSetNBPZone, OTExtractNBPName, OTExtractNBPType, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetNBPType

**FUNCTION**

OTSetNBPType — Set the NBP type portion in an NBPEntity structure.

**C INTERFACE**

```
pascal Boolean OTSetNBPType(NBPEntity* entity, const char* type);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | (x) |
| type | (x) | / |

`OTSetNBPType` will store the NBP type specified by the `type` parameter into the `NBPEntity entity`, deleting any previous type stored there. The type supplied should NOT have any of the NBP escape charaters stored in it. This function returns false if the type parameter is longer than the maximum allowed NBP type (32 characters).

**SEE ALSO**

OTSetNBPName, OTSetNBPZone, OTExtractNBPName, OTExtractNBPType, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTSetNBPZone

**FUNCTION**

OTSetNBPZone Set the NBP zone portion in an NBPEntity structure.

**C INTERFACE**

```
pascal Boolean OTSetNBPZone(NBPEntity* entity, const char* zone);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | (x) |
| zone | (x) | / |

`OTSetNBPZone` will store the NBP zone specified by the `zone` parameter into the `NBPEntity entity`, deleting any previous zone stored there. The zone supplied should NOT have any of the NBP escape charaters stored in it. This function returns false if the zone parameter is longer than the maximum allowed NBP zone (32 characters).

**SEE ALSO**

OTSetNBPName, OTSetNBPType, , OTExtractNBPName, OTExtractNBPType, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTExtractNBPName

**FUNCTION**

OTExtractNBPName Extract the NBP name from an NBPEntity structure.

**C INTERFACE**

```
pascal void OTExtractNBPName(const NBPEntity* entity, char* name);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | / |
| name | x | (x) |

`OTExtractNBPName` will extract the NBP name information from the `NBPEntity` `entity`, and store it into the string buffer specified by the `name` parameter.

**SEE ALSO**

OTSetNBPName, OTSetNBPType, OTSetNBPZone, OTExtractNBPType, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTExtractNBPType

**FUNCTION**

OTExtractNBPType Extract the NBP type from an NBPEntity structure.

**C INTERFACE**

```
pascal void OTExtractNBPType(const NBPEntity* entity, char* type);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | / |
| type | x | (x) |

`OTExtractNBPType` will extract the NBP type information from the `NBPEntity` `entity`, and store it into the string buffer specified by the `type` parameter.

**SEE ALSO**

OTSetNBPName, OTSetNBPType, OTSetNBPZone, OTExtractNBPName,, OTExtractNBPZone

# AppleTalk Utilities

A number of utility functions have been provided for those clients which do not require transport-independent behavior, and need to manipulate AppleTalk address information. They are provided to ease the transition to the Open Transport implementation of AppleTalk

## OTExtractNBPZone

**FUNCTION**

OTExtractNBPzone Extract the NBP zone from an NBPEntity structure.

**C INTERFACE**

```
pascal void OTExtractNBPZone(const NBPEntity* entity, char* zone);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| entity | (x) | / |
| zone | x | (x) |

`OTExtractNBPZone` will extract the NBP zone information from the `NBPEntity` `entity`, and store it into the string buffer specified by the `zone` parameter.

**SEE ALSO**

OTSetNBPName, OTSetNBPType, OTSetNBPZone, OTExtractNBPName, OTExtractNBPType

# Index