This chapter describes the Name-Binding Protocol (NBP) that you can use to make your process or application available to other processes or applications across the network. This chapter also describes how you can use NBP to obtain the addresses of other processes and applications on the network.

This chapter uses the term *entity* to refer to processes and applications that run on an AppleTalk network. You use NBP in conjunction with another protocol that allows you to send and receive data. For example, you can register your entity with NBP and then use a transport protocol such as ADSP to communicate with other entities; ADSP opens a socket for your entity to use and assigns that socket number to the entity. Your entity registers an NBP name in conjunction with this socket number.

You should read this chapter if you want to

- register an entity with NBP to make it available for other network entities to contact
- obtain another entity's address so that you can contact it
- obtain the NBP names and internet socket addresses of all registered entities whose NBP names match your partial specified name

For an overview of the Name-Binding Protocol and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter. For a description of the Name-Binding Protocol specification, see *Inside AppleTalk*, second edition.

About NBP

NBP allows you to bind a name to the internal storage address for your entity and register this mapping so that other entities can look it up. Applications can display NBP names to users and use addresses internally to locate entities. When you register your entity's name and address pair, NBP validates its uniqueness.

An ***entity name*** consists of three fields: ***object, type,*** and ***zone.*** The value for each of these fields can be an alphanumeric string of up to 31 characters. The entity name is not case sensitive. You specify the value for the object and type fields.

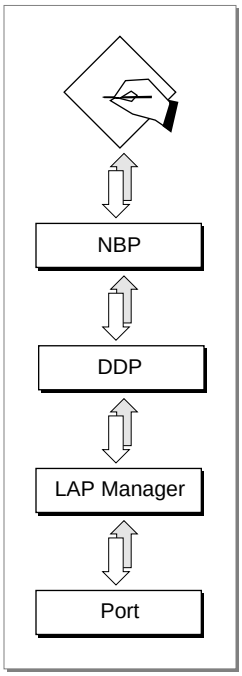
The object field typically identifies the user of the system, or the system itself, in the case of a server. Applications commonly set this value to the owner name, which the user specifies through the Sharing Setup control panel.

The type field generally identifies the type of service that the entity provides, for example, "Mailbox" for an electronic mailbox on a server. Entities of the same type can find one another and identify potential partners by looking up addresses based on the type portion of the name alone.

The zone field identifies the zone to which the node belongs. You do not specify this value; when you register your process, you specify an asterisk (*) for this field. NBP interprets the asterisk to mean the current zone or no zone, in the case of a simple network configuration not divided into zones.

The mapping of names to addresses that NBP maintains is important for AppleTalk because the addressing numbers that AppleTalk uses are not fixed. AppleTalk assigns an address dynamically to a node when the node first joins the network and whenever the node is rebooted. Because of this, the address of a node on an AppleTalk network can change from time to time. Although a network number corresponds to a particular wire and the network number portion of an address is relatively stable, the socket number that is assigned to an entity is usually randomly generated. (For an overview of AppleTalk addresses and the addressing scheme, see the chapter "Introduction to AppleTalk" in this book.) Although NBP is not a transport protocol, that is, you do not use it to send and receive data, NBP is a client of DDP. Figure 3-1 shows NBP and its underlying protocols.

Figure 3-1 The Name-Binding Protocol and the underlying AppleTalk protocols



NBP provides network entities with access to current addresses of other entities. The name part of an NBP mapping is also important in identifying and locating an entity on the network. The NBP entity name is different from the application name. An application can display entity names to users and look up addresses based on names.

For example, an entity name can include a portion that identifies that entity type. An application can request NBP to return the names of all of the registered entities of a certain type, such as a particular type of game. The application can then display those entity names to a user to allow the user to select a partner. When the user selects an entity name, the application can request NBP to return the address that is mapped to the entity name.

When you register your entity with NBP, it is made visible to other entities throughout the network. A network entity that is registered with NBP is referred to as a ***network-visible entity.*** A mail server application is an example of a network-visible entity. When a mail server is registered with NBP, workstation clients with mailboxes can access the mail server program to send and receive mail.

A server application might call NBP to register itself at initialization time so that its clients can access the server when they come online. However, a game application might register itself when a user launches it so that partner applications of the same type can locate it, then remove its entry from the NBP names directory when the user quits the application.

You use the NBP routines to register your entity so that other entities can find it and to retrieve the addresses of other entities with which you want to communicate. You specify an entity name that adheres to a defined format and register that name with NBP in conjunction with the socket number that your entity uses. NBP then makes your entity's complete address available to other entities. To retrieve the address of another entity that is registered with NBP, you supply that entity's NBP name. You can retrieve the addresses of more than one entity by using wildcards instead of a fully qualified NBP name.

Although you register your entity's NBP name in association with the socket that it uses, NBP maintains an entry that contains your entity's complete internet socket address. The ***internet socket address,*** also called the *internet address*, includes the socket number, the node ID, and the network number. All network-visible entities on an internet are ***socket clients,*** which means that each one is associated with a socket. Each socket has a unique number, and every entity has a unique internet socket address that identifies it. The socket number part of the internet address ensures that data intended for an entity is delivered to that particular entity.

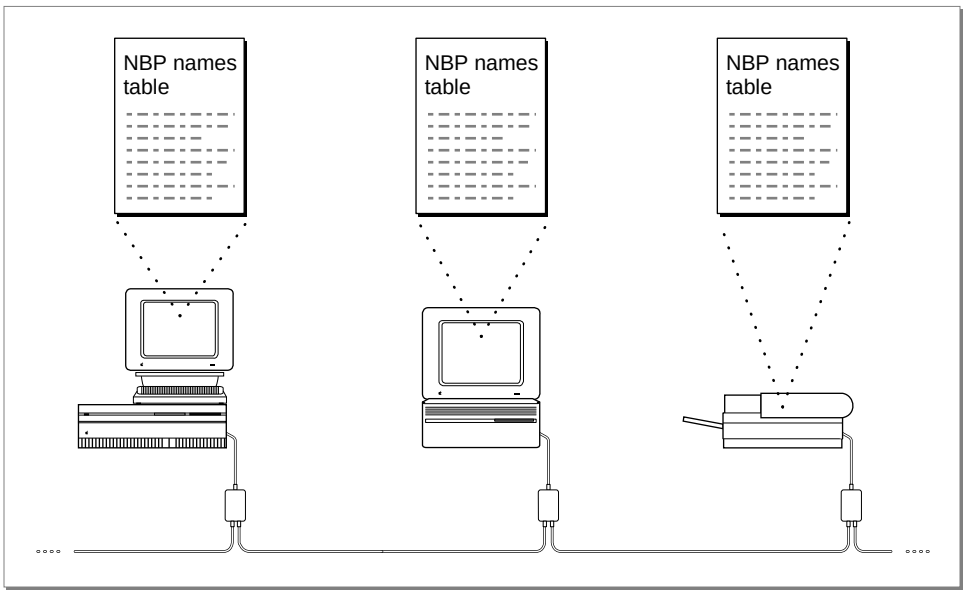
The link-access protocol dynamically assigns a unique node ID to each node when it joins the network. When the user reboots the system, sometimes the same node ID is available and sometimes a new node ID is assigned. The network number is the number of the network to which the node is directly connected, and it remains the same as long as the node is physically connected to that network. NBP fills in the node ID and the network number in a names table entry. You don't supply these parts of the internet address.

NBP maintains a ***names table*** in each node that contains the name and internet address of each registered entity in that node. Each name and address pair is called a ***tuple.*** When you register your process with NBP, you provide a names table entry. NBP builds its names table on a node from the entries that entities supply.

The NBP routines include a procedure, `NPBSetNTE`, that you can use to fill in a names table entry that is in the format that NBP expects. The `NPBSetNTE` procedure takes the name and the socket ID that you specify and builds a names table entry in the buffer that you provide. (For information on using `NPBSetNTE`, see "Registering Your Entity With NBP" beginning on page 3-7.)

To form a names table for a node, NBP connects together as a linked list the names table entries of all the registered entities on that node. The collection of names tables on all the nodes in an internet is known as the ***NBP names directory.*** Figure 3-2 shows a number of nodes on a network, each with its own names table; each names table contains an entry for each registered entity on its node.

Figure 3-2 The NBP names table on each node, collectively forming an NBP names directory



Whenever a node receives an NBP lookup request, NBP searches through its names table for a match and, if it finds a match, returns the information to the requester.

Using NBP

This section describes how you can use NBP to

- set up a names table entry for your entity and register your entity's name and address pair with NBP for other entities to access
- look up an address based on a name
- confirm a name and address that you already have
- remove your entity's name and address from the NBP names directory
- cancel a pending NBP request

The .MPP driver implements the NBP protocol. Your application should check to ensure that the .MPP driver is already loaded on the system running your application before it attempts to call NBP. If the driver is not already open, your application should open it by

calling the Device Manager's `OpenDriver` function. The following example shows how to open the .MPP driver.

```
BEGIN
   myErr := OpenDriver('.MPP', mppRefNum);   {open .MPP driver}
   IF myErr <> noErr THEN DoErr(myErr);      {check and handle }
                                             { error}
```

For more information on determining if the .MPP driver is open and opening the AppleTalk drivers, see the chapter "AppleTalk Utilities" in this book.

Your application can have multiple concurrent active NBP requests. For example, your application can perform a number of `PRegisterName`, `PLookupName` and `PConfirmName` requests concurrently. The maximum number of concurrent requests is machine dependent. You can use the `PGetAppleTalkInfo` function to determine the maximum number of concurrent NBP requests supported by the .MPP driver on the node running your application. For information about the `PGetAppleTalkInfo` function, see the chapter "AppleTalk Utilities" in this book.

All of the NBP functions use parameter blocks to hold input and output values. Whether you execute a function synchronously or asynchronously, you must not alter the contents of the parameter block until after the NBP function that uses it completes the operation. In effect, the parameter block belongs to the NBP function until the function completes execution. (For a discussion of synchronous and asynchronous execution, see the chapter "Introduction to AppleTalk" in this book.) When the operation completes, you can either reuse the memory allocated for the parameter block or release it.

In addition to the parameter block used for the function, the memory that you allocate for any records and buffers whose pointers you pass to NBP through a parameter block field must also be nonrelocatable until the function completes execution. When the operation completes, you can reuse these data structures or release the memory that you allocated for them.

To allocate nonrelocatable memory, you can use the Memory Manager's `NewPtr` or `NewPtrSys` function. If you use `NewHandle` instead, you need to lock the memory. For more information about these functions, see *Inside Macintosh: Memory.*

Registering Your Entity With NBP

You register your entity with NBP to make its services available to other entities throughout the network. Once the entity is registered, other entities can look up its name and address pair based on its name or a part of that name.

Your process can register itself with several names all associated with the same socket.

To register itself, your entity calls two NBP routines:

- the set names table entry (`NBPSetNTE`) procedure, which prepares the names table entry
- the register name (`PRegisterName`) function, which provides NBP with a pointer to the names table entry so that NBP can register the entry on the node

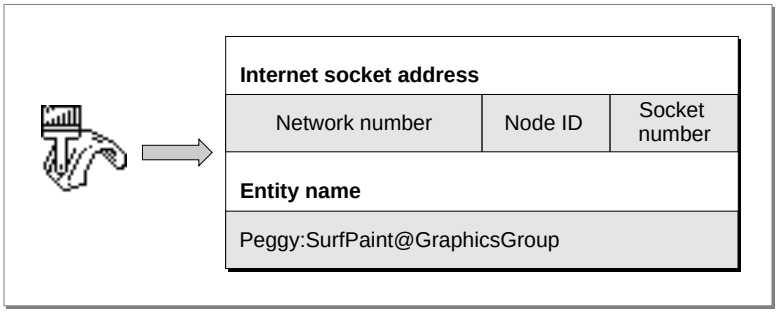
Setting Up a Names Table Entry

The `NBPSetNTE` procedure creates a names table entry in the format that Figure 3-4 on page 3-9 shows. You associate an NBP entity name with the socket number assigned to your entity.

When you create the names table entry, you provide NBP with the socket number that your entity uses. This is the socket ID that was assigned to your entity when it opened a socket.

Figure 3-3 shows a complete internet socket address belonging to an entity and the entity name that is associated with the address.

Figure 3-3 The internet socket address and entity name of an application

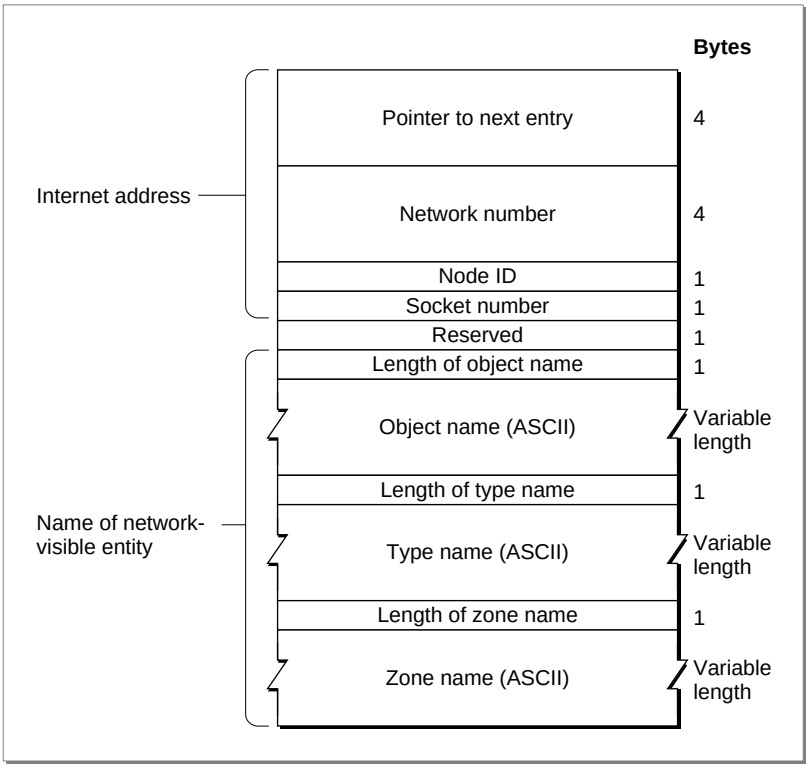


Along with the individual fields of the name and the socket number, you pass `NBPSetNTE` a pointer to a buffer that is 108 bytes long. You create a record of type `NamesTableEntry` as the buffer to be used for the names table entry. When you register your entity, NBP uses the buffer that you pass it as the actual names table entry for that entity; it does not make a copy of the buffer. NBP links the `NamesTableEntry` record that you provide to other names table entries on the node to create a names table for that node. For this reason, memory that you allocate for the buffer must be nonrelocatable.

Figure 3-4 shows the structure of the names table entry record.

Notice that the first field in the `NamesTableEntry` record is a pointer to the next entry in the linked list. NBP maintains the value of this field. You do not supply this value. However, you can get a pointer to the first entry in the names table on the node where the entity is running by calling the `PGetAppleTalkInfo` function. For information about the `PGetAppleTalkInfo` function, see the chapter "AppleTalk Utilities" in this book.

Figure 3-4 Names table entry record format



Registering a Names Table Entry

After you create the names table entry using `NBPSetNTE`, you register it by calling the `PRegisterName` function. When you call `PRegisterName`, NBP fills in the network number and node ID for the names table entry; because these values are the same for all entities on the node, you do not need to supply them.

Before you call the `PRegisterName` function, you must supply values for the function's parameter block input fields. These fields are `interval`, `count`, `entityPtr`, and `verifyFlag`. If you execute the function asynchronously, you must also supply a value for the `ioCompletion` field. After you call the `PRegisterName` function, you must not alter the contents of the parameter block until the function completes execution, and you must not modify or manipulate the names table entry until you remove it from the NBP name and address pair directory.

You set the parameter block's `entityPtr` field to the names table entry's pointer. For released software, you should always set the `verifyFlag` field to a nonzero number. This directs NBP to check throughout the network to determine that the name you want to register is unique. Ensuring that a name is unique avoids the occurrence of problems that can arise when two entities are registered with the same name. If the entity name is already registered for another entity, the `PRegisterName` function result indicates that the name is a duplicate by returning a function result of `nbpDuplicate`.

You can specify how many times NBP should attempt to verify the name's uniqueness by assigning a value to the `count` field. You can control how long NBP waits between each check by assigning a value to the `interval` field.

The interval and count parameters are both 1 byte long, which limits them to a value within the range of 0 to 255 ($00–$FF). However, you should not specify a value of 0 (which is equivalent to 256) for the retransmit interval; the task will never be executed if you do.

You measure intervals in 8-tick units. You can use this equation to determine how long in ticks a function will take to complete:

```
TimeToCompleteInTicks := count * interval * 8;
```

A value of 7 for the `interval` field is usually sufficient ($7 \times 8 = 56$ ticks equals approximately 1 second). A retry count of 5 is usually sufficient. However, on a large network, base the interval value on the speed of the network. Base the retry count on how likely it is for a particular kind of device to catch or miss the NBP lookup request and how many devices of this kind are on the network.

Some kinds of devices are more likely to receive the NBP lookup request than are others. For example, the AppleTalk ImageWriter has a dedicated processor on the LocalTalk option card to handle AppleTalk processing. A dedicated processor is likely to be available to receive an NBP lookup request, so the count for a device of this type can be relatively low. However, most Macintosh computers and LaserWriter printers depend on the system's shared processor to handle all processing, so the count for these kinds of devices should be higher. On a network with slow connections, for example, one that uses a modem bridge, you should increase the interval.

You can use different values for different types of devices. You can store these values in a preferences resource so that you can easily change them to correspond to changes in the network. For example, you could include values such as the following for these devices:

Device	Interval	Count
AppleShare	$07	$05
AppleTalk ImageWriter	$07	$02
LaserWriter	$0B	$05

You pass to the `PRegisterName` function a pointer to a parameter block and a Boolean value indicating if the function is to be executed asynchronously or synchronously. If you set the `async` Boolean parameter to `TRUE`, you must either provide a completion routine or set the `ioCompletion` field value to `NIL`, in which case, your process must poll the parameter block's `ioResult` field to determine when the function completes the operation. For a discussion of synchronous and asynchronous execution, see the chapter "Introduction to AppleTalk" in this book.

Listing 3-1 shows a segment of code that registers an application with NBP. First the code allocates nonrelocatable memory for the names table entry. Then the code calls `NBPSetNTE` to set up the names table entry in the format that the `PRegisterName` function expects.

Next, the code assigns values to the input fields of the parameter block to be used for the `PRegisterName` function. The code doesn't assign values to the `ioRefNum` and `csCode` fields because these field values are filled in by the `PRegisterName` function's glue code in the MPW interface.

Notice that the code assigns to the `entityPtr` field the `ntePtr` pointer to the buffer that the code passed to the `NBPSetNTE` function. After it sets up the parameter block, the code makes a synchronous call to the `PRegisterName` function to register the names table entry. If the `PRegisterName` function returns an error, the code releases the nonrelocatable memory that it allocated for the names table entry.

Listing 3-1 Registering an application with NBP

```
FUNCTION MyRegisterName (entityObject: Str32; entityType: Str32;
                         socket: Integer; VAR ntePtr: Ptr): OSErr;
VAR
   mppPB: MPPParamBlock;
   result: OSErr;
BEGIN
   ntePtr := NewPtrSys(sizeof(NamesTableEntry));
   IF ntePtr = NIL THEN
      BEGIN
         result := MemError;                    {return memory error}
         ntePtr := NIL;
      END
   ELSE
      BEGIN
         {Build the names table entity.}
         NBPSetNTE(ntePtr, entityObject, entityType, '*', socket);
         WITH mppPB DO
            BEGIN
               interval := $0F;                 {reasonable values for the }
               count := $03;                    { interval and retry count}
               entityPtr := ntePtr;             {pointer to NamesTableEntry}
               verifyFlag := Byte(TRUE);        {ensure that name is unique}
            END;
         result := PRegisterName(@mppPB, FALSE);{register the name}
         IF (result <> noErr) THEN
            BEGIN
               DisposPtr(ntePtr);               {if error, release memory}
               ntePtr := NIL;
            END;
      END;
   MyRegisterName := result;
END;
```

Handling Names Table Entry Requests

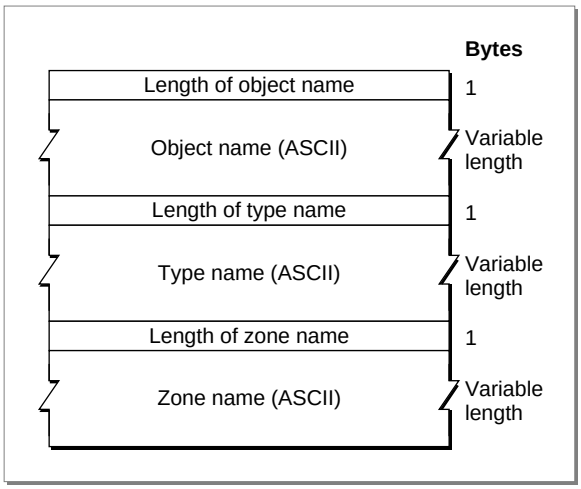
In addition to providing services that let you register an entity name and socket address for your process, NBP lets you look up addresses of other entities based on a name, confirm that a process whose entity name and address you already have is still registered with NBP and that the address is correct, remove your process's name and address from the names table when you no longer want to make the entity available, and cancel a pending request. You use

- the `NBPSetEntity` procedure to prepare an entity name in the format required by the NBP functions
- the `PLookupName` function to retrieve another entity's address based on the entity's complete NBP name, or to retrieve the addresses of multiple entities that match an NBP name that includes wildcards
- the `NBPExtract` function to read a retrieved address from the return buffer
- the `PConfirmName` function to verify a name and address
- the `PRemoveName` function to remove your process's name and address from the NBP names directory
- the `PKillNBP` function to cancel a request to register, confirm, or look up a names table entry if the function was called asynchronously and it has not already been executed

Preparing an Entity Name

To prepare an entity name using `NBPSetEntity`, you allocate a buffer that is at least 99 bytes long. You can allocate a record of type `EntityName` for this buffer. You pass `NBPSetEntity` a pointer to the buffer along with the three parts of the name (object, type, and zone), and `NBPSetEntity` writes the entity name to the buffer in the format that the `PLookupName`, `PConfirmName`, and `PRemoveName` functions require. Figure 3-5 shows the format of the entity name record.

Figure 3-5 Entity name record format



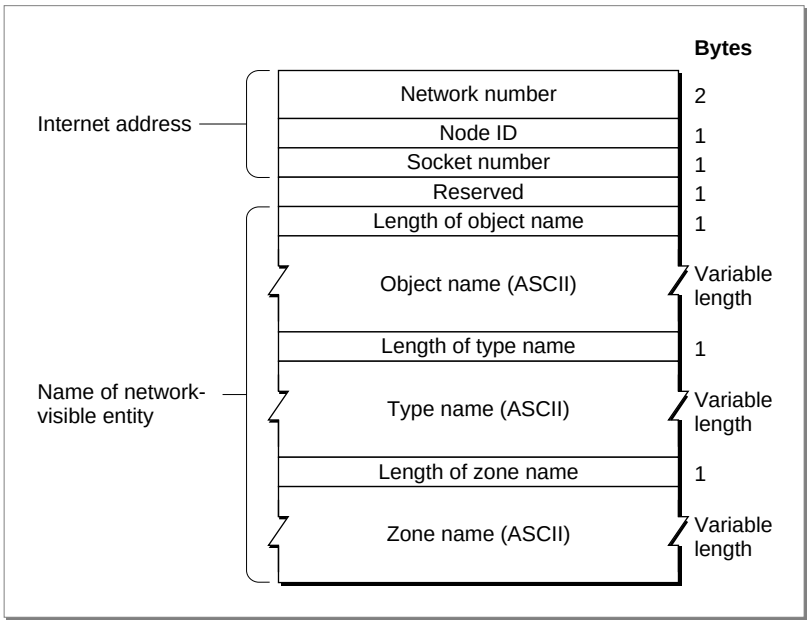
For the `PConfirmName` and `PRemoveName` functions, you must specify explicit values for the `nbpObject`, `nbpType`, and `nbpZone` parameters. However, you can specify wildcards for these parameters for `PLookupName`.

Looking Up a Name

You can use the `PLookupName` function to look up the address of a particular entity whose NBP name you know. You can also use the `PLookupName` function to find the addresses of more than one entity whose NBP names match a partial name that includes wildcards.

If you want to retrieve the address of a particular entity, you assign to the `entityPtr` field of the parameter block a pointer to a fully qualified entity name that you provided using `NBPSetEntity`. You create a buffer to hold the name and address that `PLookupName` returns and set the parameter block's return buffer pointer (`retBuffPtr`) field to this buffer's pointer. Because the data is packed and each tuple takes a maximum of 104 bytes, to look up a particular name you need to set the return buffer size (`retBuffSize`) field to the buffer size of 104 bytes. Figure 3-6 shows the format of the record for a tuple that `PLookupName` returns.

Figure 3-6 Tuple returned by the `PLookupName` function



If you want only one name and address pair returned, you set the maximum number of matches (`maxToGet`) field to 1. When you call the function asynchronously, you must assign to the `ioCompletion` field a pointer to your completion routine or set this field to `NIL`. For more information about executing routines synchronously or asynchronously, see the chapter "Introduction to AppleTalk" in this book.

If you want to obtain the addresses of other instances of the same type of entity that are running on other nodes in the network, you can look up the addresses of these entities by specifying wildcards. In this case, you specify a type field value and wildcards for the object and zone fields.

Table 3-1 shows the wildcards that you can use to control the kind of matches that you want NBP to return.

Table 3-1 NBP wildcards

Character	Meaning
=	All possible values. You can use the equal sign (=) alone instead of specifying a name in the object or type field.
≈	Any or no characters in this position. You can use the double tilde (≈) to obtain matches for object or type fields. For example, *pa≈l* matches *pal*, *paul*, *paper ball*, and so forth. You can use only one double tilde in any string. Press Option-X to type the double tilde character on a Macintosh keyboard. If you use the double tilde alone, it has the same meaning as the equal sign (=). NOTE Any node not running AppleTalk Phase 2 drivers will not recognize this character.
*	This zone. You can use the asterisk (*) in place of the name of the zone to which this node belongs.

For example, if you want to retrieve the names and addresses of all the mailboxes in the same zone as one in which your process is running, you can set the entity name object field to the equal sign (=), the type field to `Mailbox`, and the zone field to the asterisk (*). The `PLookupName` function will return the entity names and internet addresses of all mailboxes in that zone excluding your own entity's name and address.

You can specify how thorough the lookup should be by defining the number of times that NBP should broadcast the lookup packets and the time interval between these retries. To do this, you assign values to the parameter block's `count` and `interval` fields. See the discussion on how to determine these values in the section "Registering a Names Table Entry" beginning on page 3-9.

You must also create a buffer large enough to hold all of the tuples for the matches that NBP returns. (See Listing 3-3 on page 3-17.) You assign the buffer's pointer to the parameter block's `retBuffPtr` field and the buffer's size in bytes to the `retBuffSize` field. Allow 104 bytes for each match. You set the maximum number of matches for NBP to return as the value of the `maxToGet` field.

The `PLookupName` function keeps track of the number of matches it writes to the return buffer each time it receives a returned packet containing one or more matches, and it updates the number of matches returned (`numGotten`) field after it returns each match. Because `PLookupName` maintains `numGotten`, you can start reading the names and addresses in the buffer and storing them or displaying them for the user before the function completes execution.

A single lookup request or retry can return more than one match in a reply packet. When this happens, the `PLookupName` function will return as many of the matches that the packet contains as will fit in the buffer. In cases such as this, you will find that the number of tuples that `PLookupName` writes to the return buffer may exceed the maximum number of matches to be returned as specified by `maxToGet`. When this occurs you can assume that there may be additional matches that did not fit in the buffer or additional reply packets containing matches that `PLookupName` did not process. To receive all the matches, you should increase the size of the buffer and the `maxToGet` number, and call the `PLookupName` function again.

If the buffer is too small to accommodate all of the returned matches in a packet, the `PLookupName` function returns a function result of `nbpBuffOvr`. In any case, the `numGotten` field always indicates the actual number of tuples returned in the buffer. (See also "PLookupName" beginning on page 3-30 for more information about this function.)

The code in Listing 3-2 assigns values to the fields of the parameter block to be used for the `PLookupName` function call. The value `theEntity` points to a packed entity-name record that you prepared using `NBPSetEntity`. This is the name that will be looked for. The value `returnBufferPtr` points to the buffer where `PLookupName` will return any matches that it finds. The buffer must be able to hold the number of matches specified by the input value of `entityCount`; each match is 104 bytes long. On return, `entityCount` contains the number of matches that the `PLookupName` function found and returned in the buffer pointed to by `returnBufferPtr`. The `PLookupName` function's glue code in the MPW interface fills in the values for the `ioRefNum` and `csCode` fields.

Listing 3-2 Calling `PLookupName` to find matches for an entity name

```
FUNCTION MyLookupName (theEntity: EntityName; VAR entityCount: Integer;
                       returnBufferPtr: Ptr): OSErr;
CONST
   kTupleSize = 104;       {sizeof(AddrBlock) + a one-byte enumerator + }
                           { sizeof(EntityName)}
VAR
   mppPB: MPPParamBlock;

BEGIN
   WITH mppPB DO
      BEGIN
         interval := $0F;                {reasonable values for the }
         count := $03;                   { interval and retry count}
         entityPtr := @theEntity;        {pointer to the entity name to }
                                         { look for}
         retBuffPtr := returnBufferPtr;  {pointer to the buffer for the }
                                         { tuples}
```

```
        RetBuffSize := entityCount * kTupleSize;
                                        {return buffer size}
        maxToGet := entityCount;        {the number of entities that the }
                                        { return buffer can hold}
      END;
   MyLookupName := PLookupName(@mppPB, FALSE);
                                {look up the entity name}
   entityCount := mppPB.numGotten;
                                {return the number of matches found}
END;
```

The tuples in the buffer are in the format used in the NBP names table, as shown in Figure 3-6 on page 3-13. Because data is packed, the object, type, and zone names in this format are of arbitrary length; you cannot use Pascal to read these tuples. You can use the `NBPExtract` function to read tuples from the buffer.

Extracting a Name From a List of Returned Names

After NBP returns the matches to your buffer, you need to extract the match or matches that you want to use. You can use the `NBPExtract` function to read a name and address pair from the return buffer that you supplied to `PLookupName`. Before you call `NBPExtract`, you need to allocate memory for two buffers: one buffer that is at least 102 bytes long to hold the name part of the tuple and another buffer that is 4 bytes long to hold the address. You pass the `NBPExtract` function pointers to these buffers. The `NBPExtract` function unpacks the name and address data and writes it to the buffers that you supply.

You also pass `NBPExtract` a pointer to the buffer containing the returned tuples; this is the pointer that you assigned to the `PLookupName` function's `retBuffPtr` parameter block field. For the `numInBuf` parameter, you specify the number of tuples in the return buffer; this is the value that the `PLookupName` function returned in the `numGotten` parameter block field. Counting the first returned tuple as one and following in sequence to the value of `numGotten`, you identify which name and address pair you want to extract as the value of the `whichOne` parameter. You can use the `NBPExtract` function in a loop that varies the value of the `whichOne` parameter (`entityCount` in the following code example) from 1 to the total number of tuples in the list to extract all the names in the list.

Listing 3-3 shows an application-defined procedure, `DoMyLookupName`, that allocates a buffer to hold the matches that the `PLookupName` function returns; the `MyLookupName` function, shown in Listing 3-2 on page 3-15, calls the `PLookupName` function. The `DoMyLookupName` procedure calls the `MyLookupName` function.

If the `MyLookupName` function returns a result code of `noErr`, then the code calls the `NBPExtract` function to read the matches that are in the buffer and write them to the application's buffer with an application-defined routine, `MyAddToMatchList`; the listing does not show the `MyAddToMatchList` routine. After the matches are extracted, the code disposes of the return buffer.

Listing 3-3 Creating a buffer to hold name matches found, then using `NBPExtract` to read the matches

```
PROCEDURE DoMyLookupName;
   CONST
      kTupleSize = 104;       {sizeof(AddrBlock) + a one-byte enumerator + }
                              { sizeof(EntityName)}
      kMaxMatches = 100;      {number of matches to get}
   VAR
      result:           OSErr;
      returnBufferPtr:  Ptr;
      theEntity:        EntityName;
      entityCount:      Integer;
      index:            Integer;
      entityAddress:    AddrBlock;
BEGIN
   returnBufferPtr := NewPtr(kMaxMatches * LongInt(kTupleSize));
   IF returnBufferPtr <> NIL THEN
      BEGIN
      {Create a packed entity name.}
         NBPSetEntity(@theEntity, '=', 'AFPServer', '*');
         entityCount := kMaxMatches;      {maximum number of matches we want}
         result := MyLookupName(theEntity, entityCount, returnBufferPtr);
         IF result = noErr THEN
         {Extract the matches and add them to the match list.}
            FOR index := 1 TO entityCount DO
               IF NBPExtract(returnBufferPtr, entityCount, index, theEntity,
                             entityAddress) = noErr THEN
                  AddToMatchList(theEntity, entityAddress)
            DiposPtr(returnBufferPtr);    {release the memory}
   END;
END;
```

Confirming a Name

If you know the name and address of an entity, and you only want to confirm that the tuple is still registered with NBP and that the address hasn't been changed, you should call the `PConfirmName` function instead of calling `PLookupName`.

The `PConfirmName` function is faster than `PLookupName` because NBP can send a request packet directly to the node based on the address that you supply rather than having to broadcast lookup packets throughout the network to locate the names table entry based on the entity name alone.

The code in Listing 3-4 sets up the parameter block to be used for the `PConfirmName` function and calls `PConfirmName` to verify that the name and address still exist, and

that the address is unchanged. If the application is using a different socket, PConfirmName returns a function result of nbpConfDiff and gives the new socket number in the parameter block's newSocket field.

Listing 3-4 Confirming an existing NBP name and address

```
FUNCTION MyConfirmName (theEntity: EntityName; entityAddress: AddrBlock;
                        VAR socket: Integer): OSErr;
VAR
   mppPB: MPPParamBlock;
BEGIN
   WITH mppPB DO
      BEGIN
         interval := $0F;              {reasonable values for the interval }
         count := $03;                 { and retry count}
         entityPtr := @theEntity;      {entity name to look for}
         confirmAddr := entityAddress; {entity's network address}
      END;
   MyConfirmName := PConfirmName(@mppPB, FALSE);
   socket := mppPB.newSocket;          {return the socket number, which is }
                                       { the new socket number if }
                                       { PConfirmName's result is }
                                       { nbpConfDiff}
END;
```

Removing an Entry From the Names Table

After you close the socket that your process uses or when you no longer want to make the process available throughout the network, you remove the names table entry from the node on which it resides using the PRemoveName function.

There are two ways to remove a names table entry:

- For the first method, you use the NBPSetEntity procedure to put the entity name of an existing registered entity into the structure that NBP requires. Then you specify the pointer to this record as the value of the entityPtr field of the parameter block.
- For the second method, you provide the PRemoveName function with a pointer to the names table entry record that you used to register the name.

The PRemoveName function removes the entry from the node's names table unless the name is no longer registered, in which case, PRemoveName returns a function result of nbpNotFound. An entity name may not be included in the node's names table if, for example, the request to register the name had been canceled by the PKillNBP function before the PRegisterName function used to register the name was executed.

The code in Listing 3-5 shows how to remove a names table entry using PRemoveName. The PRemoveName function's glue code fills in the ioRefNum and csCode values. The code in Listing 3-5 provides the pointer to the names table entry record that was used to

register the name; it assigns this value to the `entityPtr` field of the parameter block used for the `PRemoveName` function call. (The code in Listing 3-1 on page 3-11 created the names table entry record.) If the application-defined `MyRemoveName` function returns a function result of `noErr`, the code disposes of the memory block pointed to by `ntePtr`.

Listing 3-5 Removing an NBP names table entry

```
FUNCTION MyRemoveName (ntePtr: Ptr): OSErr;
VAR
   mppPB: MPPParamBlock;
   result: OSErr;
BEGIN
   mppPB.entityPtr := Ptr(ORD4(ntePtr) + 9);
                    {the entity name is at offset 9 in the NTE}
   result := PRemoveName(@mppPB, FALSE);{remove the name}
   IF (result = noErr) THEN
      DisposPtr(ntePtr); {release the memory}
   MyRemoveName := result;
END;
```

Canceling a Request

You can use the `PKillNBP` function to cancel a request to register, look up, or confirm a names table entry if the function was called asynchronously and it has not already been executed.

When you call `PRegisterName`, `PLookupName`, or `PConfirmName`, NBP calls the Device Manager, which places your request in the .MPP driver queue with other requests waiting to be executed. To queue the request, the Device Manager places a pointer to the function's parameter block in the .MPP driver queue. You assign this pointer to the `PKillNPB` parameter block's queue element (`nKillQEl`) field.

If the function request that you want to cancel is not in the queue, `PKillNBP` returns a function result of `cbNotFound`. If `PKillNBP` cancels the function, it returns a function result of `noErr`, and the function that it canceled returns a function result of `reqAborted`.

The code in Listing 3-6 on page 3-20 shows how to cancel a `PRegisterName`, `PLookupName`, or `PConfirmName` function call. The application-defined `MyKillNBP` function takes as an input parameter a pointer to the parameter block that was used to make the `PLookupName`, `PRegisterName`, or `PConfirmName` function call to be canceled. The code assigns this pointer to the `nKillQEl` field of the parameter block to be passed to the `PKillNBP` function. The `ioRefNum` and `csCode` field values are filled in by the `PKillNBP` function's glue code in the MPW interface.

Listing 3-6 Canceling a request to look up a name

```
FUNCTION MyKillNBP (requestPBPtr: MPPPBptr): OSErr;
   VAR
   mppPB: MPPParamBlock;
BEGIN
   mppPB.nKillQEl := Ptr(requestPBPtr);
   MyKillNBP := PKillNBP(@mppPB, FALSE);
END;
```

NBP Reference

This section describes the data structures and routines that are specific to the Name-Binding Protocol (NBP). The "Data Structures" section shows the Pascal data structures for the records and the parameter block that the NBP functions use. The "Routines" section describes the NBP routines.

Data Structures

This section describes the data structures that you use to provide information to and receive it from NBP.

Address Block Record

The address block record is a data structure of type `AddrBlock` that defines a packed record that is used to contain an internet socket address. The names table entry record includes a field that takes a value of this record type.

```
AddrBlock = PACKED RECORD
   aNet:       Integer;
   aNode:      Byte;
   aSocket:    Byte;
END;
```

Field descriptions

`aNet` The network number.

`aNode` The node ID.

`aSocket` The socket number.

Names Table Entry Record

The names table entry record is a data structure of type `NamesTableEntry` that is used to hold an NBP names table tuple, consisting of a name and address. Because the object, type, and zone names in a names table entry are packed data of arbitrary length, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). If you are using the NBP Pascal interface, you use the `NPBSetNTE` procedure to create a names table entry. For illustration of the names table record format, see Figure 3-4 on page 3-9.

```
TYPE
   NamesTableEntry =
   RECORD
      qLink:      QElemPtr;
      nteAddress: AddrBlock;
      nteData: PACKED ARRAY[1..100] OF Char;
   END;
```

Field descriptions

`qLink`	A pointer to the next names table entry in the names table linked list that NBP maintains on the node. (This field is used internally by NBP.)
`nteAddress`	The internet socket address.
`nteData`	The NBP name associated with the entity's address.

Entity Name Record

The entity name record is a data structure of type `EntityName` that is used to hold the NBP name for an entity that is associated with a socket address. Your application looks up or confirms an address or removes a names table entry based on an entity name.

Because the object, type, and zone names that constitute the entity name in this format are packed data and of arbitrary length, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). If you are using the NBP Pascal interface, you put an existing entity name into the structure that NBP requires using the `NBPSetEntity` procedure.

```
TYPE
   EntityName =
   RECORD
      objStr:  Str32;
      typeStr: Str32;
      zoneStr: Str32;
   END;
   EntityPtr = ^EntityName;
```

Field descriptions

objStr	The object part of an entity name. It consists of an alphanumeric string of up to 31 characters. The object part of the name can be any valid string; it is commonly used to identify the user of the system.
typeStr	The type part of an entity name. It consists of an alphanumeric string of up to 31 characters. The type part of the name can be any valid string, but it is commonly used to identify the type of service that the entity provides.
zoneStr	The zone part of an entity name. It consists of an alphanumeric string of up to 31 characters that identifies the zone to which the node belongs that is running the process.

The MPP Parameter Block for NBP

The NBP functions use the MPP parameter block defined by the `MPPParamBlock` data type to pass information to and receive it from the .MPP driver. You use these fields to specify input values to and receive output values from an NBP function. This section defines the fields common to all NBP functions, except those that are reserved for internal use by the .MPP driver or not used.

```
TYPE
   MPPParmType    =     (...RegisterNameParm, LookupNameParm,
                        ConfirmNameParm,RemoveNameParm, KillNBPParm...);
   MPPPBPtr       =     ^MPPParamBlock;
   MPPParamBlock  =
   PACKED RECORD
      qLink:               QElemPtr;      {reserved}
      qType:               Integer;       {reserved}
      ioTrap:              Integer;       {reserved}
      ioCmdAddr:           Ptr;           {reserved}
      ioCompletion:        ProcPtr;       {completion routine}
      ioResult:            OSErr;         {result code}
      ioNamePtr:           StringPtr;     {reserved}
      ioVRefNum:           Integer;       {reserved}
      ioRefNum:            Integer;       {driver reference number}
      csCode:              Integer;       {primary command code}
   CASE MPPParmType OF
      RegisterNameParm,
      LookupNameParm,
      ConfirmNameParm,
      RemoveNameParm:
         (interval:        Byte;          {retry interval}
         count:            Byte;          {retry count}
         entityPtr:        Ptr;           {pointer to entity name or }
                                          { names table element}
```

```
      CASE MPPParmType OF
         RegisterNameParm:
            (verifyFlag:   Byte;       {verify uniqueness of name or not}
            filler3:       Byte;)
         LookupNameParm:
            (retBuffPtr:   Ptr;        {pointer to return buffer}
            retBuffSize:   Integer;    {return buffer size}
            maxToGet:      Integer;    {matches to get}
            numGotten:     Integer;)   {matches gotten}
         ConfirmNameParm:
            (confirmAddr:  AddrBlock;  {pointer to entity name}
            newSocket:     Byte;       {socket number}
            filler4:       Byte);
         )
   KillNBPParm:
      (nKillQEl:        Ptr;)       {pointer to queue element to cancel}
END;
```

The fields for each variant record are defined in the function description that uses the record.

Routines

This section describes the NBP routines. The NBP routines allow you to

- create an NBP names table entry
- register an NBP names table entry with the NBP names directory
- put an existing NBP entity name into the structure that NBP requires for you to look up, confirm, or remove an existing registered entity name
- look up the address of a network entity based on its NBP name
- read a name and address from a list of pairs that NBP returns
- confirm that a name and address pair is registered with NBP
- remove a registered name from the NBP names directory
- cancel an NBP request

An arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both:

Arrow	Meaning
→	Input
←	Output
↔	Both

You can use the `PGetAppleTalkInfo` function to determine the maximum number of concurrent NBP requests that the .MPP driver installed on the system that is running your process supports. See the chapter "AppleTalk Utilities" for information on the `PGetAppleTalkInfo` function.

Registering an Entity

This section describes the `NBPSetNTE` and the `PRegisterName` routines. You can use the `NBPSetNTE` procedure to create an NBP names table entry to be used to register the name and address of an entity with NBP so that the entity is made visible throughout the network. You use the `PRegisterName` function to register a names table entry that you created through the `NBPSetNTE` procedure.

NBPSetNTE

The `NBPSetNTE` procedure creates a new NBP names table entry to be added to the NBP names table through the `PRegisterName` function.

```
PROCEDURE NBPSetNTE (ntePtr: Ptr; nbpObject,nbpType,nbpZone: Str32;
                     socket: Integer);
```

`ntePtr` A pointer to a buffer that you provide that is at least 108 bytes long. The `NBPSetNTE` procedure fills this buffer with a names table entry based on the remaining parameter values that you specify. This buffer should be a record of type `NamesTableEntry`.

`nbpObject` The object part of the name for the names table entry. This value can be up to 31 characters long. You cannot use any wildcard characters in this name. (An object name typically identifies the node and is commonly set to the Chooser name that the user specified.)

`nbpType` The type part of the name for the names table entry. This value can be up to 31 characters long. You cannot use any wildcard characters in this name. This part of an NBP name usually identifies the type of service to which the name is assigned.

`nbpZone` The zone part of the name for the names table entry. You must use an asterisk (*) for this name, indicating the local zone.

`socket` The number of the socket that was returned and assigned to your process when you opened a socket using one of the AppleTalk transport protocols. The NBP entity name is associated with the socket number that you specify.

DESCRIPTION

The `NBPSetNTE` procedure creates a names table entry that you can register with the NBP names directory using the `PRegisterName` function. When you call `PRegisterName` to register the name, you must provide a pointer to the NBP names table entry that you created previously.

Because the object, type, and zone names in a names table entry are packed data of arbitrary length, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). Use the `NBPSetNTE` procedure to create the names table entry.

SPECIAL CONSIDERATIONS

The names table entry that you provide remains the property of NBP once you register it using `PRegisterName` and until you remove it using the `PRemoveName` function. You can allocate a block of nonrelocatable memory for the names table entry buffer using the Memory Manager's `NewPtr` or `NewPtrSys` function.

If instead you use the `NewHandle` function to allocate the buffer memory, you must lock the memory before you call `PRegisterName` to register the name because NBP adds the actual names table entry to the NBP names table for that node, and the names table entry remains part of the table until you remove it.

ASSEMBLY-LANGUAGE INFORMATION

The `NBPSetNTE` procedure is implemented entirely in the MPW interface files. There is no assembly-language equivalent for this procedure.

SEE ALSO

For the names table entry record format, see Figure 3-4 on page 3-9.

For the `NamesTableEntry` data type declaration, see "Data Structures" on page 3-20.

For information on allocating memory, see *Inside Macintosh: Memory.*

The `PRegisterName` function is described next.

PRegisterName

The `PRegisterName` function adds a unique names table entry to the local node's NBP names table.

```
FUNCTION PRegisterName (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr`	A pointer to an MPP parameter block.
`async`	A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	ioRefNum	Integer	The .MPP driver reference number.
→	csCode	Integer	Always registerName.
→	interval	Byte	The retry interval.
↔	count	Byte	The retry count.
→	entityPtr	Ptr	A pointer to a names table entry.
→	verifyFlag	Byte	A flag to indicate whether NBP is to verify NBP names as unique.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .MPP driver calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .MPP driver ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum` The .MPP driver reference number. The MPW interface fills in this field.

`csCode` The command code of the .MPP command to be executed. The MPW interface fills in this field.

`interval` The retry interval to be used by NBP when it verifies the uniqueness of the name. The retry interval value specifies how long the function is to wait between retries in 8-tick units. A value of 7 for the `interval` field is usually sufficient ($7 \times 8 = 56$ ticks equals approximately 1 second).

`count` On input, the retry count to be used by NBP when it verifies the uniqueness of the name. Its value tells the `PRegisterName` function how many times to retry. A retry count of 5 is usually sufficient. On return, the number of times that NBP actually attempted to verify the uniqueness of the name.

`entityPtr` A pointer to a names table entry. You can use the `NBPSetNTE` procedure to create a names table entry.You cannot use wildcard characters in the object name and type name fields of the names table entry, but you must use an asterisk (*)—indicating the local zone—for the zone name field.

`verifyFlag` A flag that determines whether NBP attempts to verify that the name you are adding to the names table is unique. Set this flag to a nonzero number to have NBP verify the name. You can set this flag to zero during program development, but to avoid confusion caused by duplicate names on a network, you should always set the `verifyFlag` parameter to a nonzero number in released software.

DESCRIPTION

Before another entity can send information to your entity over AppleTalk, it must have your entity's internet socket address. Also, for users to be able to select your application, the entity must be made visible throughout the network.

The `PRegisterName` function adds an entry for a network entity to the node's NBP names table, making it possible for a user or another process to locate that entity through its NBP name (consisting of object, type, and zone names). The process whose name is registered with NBP is referred to as a *network-visible entity.*

Because the object, type, and zone names in a names table entry are of arbitrary length, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). Use the `NBPSetNTE` procedure to create the names table entry. If you execute the function asynchronously and you do not specify a completion routine, your process can poll the `ioResult` field to determine when the function completes execution.

You can assign any number of names to a single socket. If you use a single socket for more than one process, you must provide a socket listener.

If you use the `PKillNPB` function to cancel the `PRegisterName` function and the cancel request is successful, `PRegisterName` returns a function result of `reqAborted`.

SPECIAL CONSIDERATIONS

The names table entry that you provide remains the property of NBP until you use the `PRemoveName` function to remove the entry from the names table. You must allocate a nonrelocatable block for the names table entry, or lock any relocatable block that you use for it until you are ready to remove the entry.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PRegisterName` function from assembly language, call the `_Control` trap macro with a value of `registerName` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. To execute this function from assembly language, you must also specify the driver reference number.

RESULT CODES

`noErr`	0	No error
`nbpDuplicate`	–1027	Name already exists
`tooManyReqs`	–1097	Too many concurrent requests; wait a few minutes, then try the request again
`reqAborted`	–1105	Request canceled

SEE ALSO

To create a names table entry, use the `NBPSetNTE` procedure, described on page 3-24.

For the names table entry record format, see Figure 3-2 on page 3-6.

For the `NamesTableEntry` data type declaration, see "Names Table Entry Record" on page 3-21.

To cancel a name registration request, use the `PKillNBP` function, described on page 3-38.

For information about socket listeners, see the chapter "Datagram Delivery Protocol (DDP)" in this book.

Handling Name and Address Requests

This section describes

- the `NBPSetEntity` procedure, which you can use to put an existing NBP entity name into the structure that NBP requires for you to look up, confirm, or remove an existing registered entity name
- the `PLookupName` function, which you can use to look up the network address of an entity, based on the NBP registered name for that entity, or using wildcards
- the `NBPExtract` function, which you can use to read a name and address pair from the buffer containing the list of tuples that `PLookupName` returns
- the `PConfirmName` function, which you can use to confirm that a name whose address you know is still associated with that address, and that the pair is still registered with the NBP names directory
- the `PRemoveName` function, which you can use to remove a name and address pair from the NBP names directory when you no longer want to make the service associated with the tuple available throughout the network
- the `PKillNBP` function, which you can use to cancel requests to NBP

NBPSetEntity

The `NBPSetEntity` procedure puts an existing NBP name of a network-visible entity into the packed-record format that the `PLookupName`, `PConfirmName`, and `PRemoveName` functions require.

```
PROCEDURE NBPSetEntity (buffer: Ptr;
                        nbpObject,nbpType,nbpZone: Str32);
```

`buffer` A pointer to a buffer that you provide that is at least 99 bytes long. The `NBPSetEntity` procedure fills this buffer with the entity name you specify in the other three parameters.

`nbpObject` The object part of the registered NBP name. You can specify wildcard characters in this part of the name only for use with the `PLookupName` function.

`nbpType` The type part of the registered NBP name. You can use wildcard characters in this part of the name only for use with the `PLookupName` function.

`nbpZone` The zone part of the registered NBP name. You can use wildcard characters in this part of the name only for use with the `PLookupName` function.

Table 3-1 on page 3-14 describes the wildcard characters that you can specify for the `nbpObject`, `nbpType`, and `nbpZone` fields for use with the `PLookupName` function.

DESCRIPTION

When you call the `PRemoveName` function to remove the name of a network-visible entity from the NBP names table, or call the `PLookupName` or `PConfirmName` function to look up network-visible entities, you must specify an entity name in the format shown in Figure 3-5 on page 3-12. (For `PRemoveName`, instead of creating the entity-name record, you can provide a pointer to the names table entry record that you used to register the name.)

The object, type, and zone names that constitute the entity name in this format are packed data and of arbitrary length. Therefore, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). Use the `NBPSetEntity` procedure to provide the entity name in the format that NBP requires.

SPECIAL CONSIDERATIONS

The memory that you allocate for the entity name buffer belongs to NBP until the function completes execution. You can reuse it or dispose of it after the operation completes.

ASSEMBLY-LANGUAGE INFORMATION

The `NBPSetEntity` procedure is implemented entirely in the MPW interface files. There is no assembly-language equivalent for this procedure.

SEE ALSO

The `PLookupName` function is described next.

For a discussion of how to use `NBPSetEntity`, see "Preparing an Entity Name" beginning on page 3-12.

To confirm that an entity is still registered with NBP, use the `PConfirmName` function, described on page 3-34.

To remove a registered name from the NBP names table, use the `PRemoveName` function, described on page 3-36.

PLookupName

The `PLookupName` function returns the names and addresses of all the network-visible entities that match a name that you supply, which can include wildcard characters.

```
FUNCTION PLookupName (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioRefNum`	`Integer`	The .MPP driver reference number.
→	`csCode`	`Integer`	Always `lookupName`.
→	`interval`	`Byte`	The retry interval.
↔	`count`	`Byte`	The retry count.
→	`entityPtr`	`Ptr`	A pointer to an entity name.
→	`retBuffPtr`	`Ptr`	A pointer to the return data buffer.
→	`retBuffSize`	`Integer`	The return buffer size in bytes.
→	`maxToGet`	`Integer`	The maximum number of matches to get.
←	`numGotten`	`Integer`	The number of addresses found and returned.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .MPP driver calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .MPP driver ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum` The .MPP driver reference number. The MPW interface fills in this field.

`csCode` The command code of the .MPP command to be executed. The MPW interface fills in this field.

`interval` The retry interval to be used by NBP when it looks on the internet for matching names. The retry interval value specifies how long the function is to wait between retries in 8-tick units. The retry interval equals the `interval` field value × 8 ticks. A value of 7 for the

	`interval` field is usually sufficient ($7 \times 8 = 56$ ticks equals approximately 1 second). However, on a large network, you should base the interval value on the speed of the network and how many devices of this type you expect to be on the network.
`count`	The retry count to be used by NBP when it looks on the internet for matching names. Its value specifies the number of times `PLookupName` is to retry the operation. A retry count of 3 or 4 is usually sufficient. However, on a large network, you should base the value on how likely it is for the type of device to miss the NBP request. For example, the AppleTalk ImageWriter has a dedicated processor on the LocalTalk option card to handle AppleTalk processing, so the retry count for a device of this type can be low, whereas most Macintosh systems and LaserWriter printers depend on their shared processor to handle all system processing, so a retry count for a device of these types should be higher. The `PLookupName` function decrements this field each time it looks for names.
`entityPtr`	A pointer to an entity name in the format shown in Figure 3-5 on page 3-12. You can use the `NBPSetEntity` procedure to prepare the entity name record.
`retBuffPtr`	A pointer to a buffer you provide into which the `PLookupName` function puts the names and addresses that it finds. Each matching tuple takes a maximum of 104 bytes, and you use the `maxToGet` field to specify the maximum number of tuples to be returned.
`retBuffSize`	The size of the buffer you are providing.
`maxToGet`	The maximum number of matches to be returned.
`numGotten`	The actual number of matches that `PLookupName` returned. The `PLookupName` function updates this field each time it receives an NBP returned packet and adds names to the return buffer. If there is space remaining in the buffer, NBP may return more matches than the number specified by `maxToGet`. If `numGotten` is greater than or equal to `maxToGet`, there may be additional matches. In this case, you should increase the size of the buffer pointed to by `retBuffPtr` and call the `PLookupName` function again.

DESCRIPTION

Before you can send data to another entity, you must have the network address of that entity. The `PLookupName` function returns the names and addresses of any network-visible entities whose names match the entity name you specify. The entity name can include any of the wildcard characters given in Table 3-1 on page 3-14.

The `PLookupName` function completes execution when the number of matches returned is equal to or greater than the number in the `maxToGet` field, the function exceeds the retry count, the buffer overflows, or the request is canceled through the `PKillNBP` function.

The number of matches returned can be greater than the number specified in the `maxToGet` field under the following circumstances: A single lookup request or retry can return more than one match in a reply packet. If there is space remaining in the buffer

and NBP receives a packet containing multiple matches, `PLookupName` will return as many of the matches as fit in the buffer. If this occurs, you should increase the size of the buffer and call the `PLookupName` function again to ensure that you obtain all of the matches.

If all of the tuples returned in a reply packet do not fit in the buffer, then the function completes with as many tuples as can fit. Whether NBP returns more or fewer matches than you specify as the value of `maxToGet`, the value of `numGotten` reflects the actual number of tuples that `PLookupName` writes to the return buffer.

Because the function updates the `numGotten` field each time it receives a returned packet containing one or more matches and writes those name and address pairs to the return buffer, you can start reading the names in the buffer and displaying them for the user before the function completes execution.

The tuples in the buffer are in the format used in the NBP names table, as shown in Figure 3-6 on page 3-13. Because the object, type, and zone names in this format are of arbitrary length, you cannot use Pascal to read these tuples. Use the `NBPExtract` function to read tuples from the buffer.

SPECIAL CONSIDERATIONS

Memory used for the entity name record and the return buffer belongs to `PLookupName` until the function completes execution and must be nonrelocatable.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PLookupName` function from assembly language, call the `_Control` trap macro with a value of `lookupName` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. To execute this function from assembly language, you must also specify the driver reference number.

RESULT CODES

`noErr`	0	No error
`tooManyReqs`	–1097	Too many concurrent requests; wait a few minutes, then try the request again
`reqAborted`	–1105	Request canceled

SEE ALSO

To read tuples from the buffer, use the `NBPExtract` function, described next.

To create the entity name record, use the `NBPSetEntity` procedure, described on page 3-28.

To check that a network-visible entity whose name and address you already know is still available on the network, use the `PConfirmName` function, described on page 3-34.

To cancel a name lookup request, use the `PKillNBP` function, described on page 3-38.

NBPExtract

The `NBPExtract` function returns one tuple (entity name and internet address) from the list of tuples placed in a buffer by the `PLookupName` function.

```
FUNCTION NBPExtract (theBuffer: Ptr; numInBuf: Integer;
                     whichOne: Integer;
                     VAR abEntity: EntityName;
                     VAR address: AddrBlock): OSErr;
```

`theBuffer` A pointer to the buffer containing the tuples returned by the `PLookupName` function.

`numInBuf` The number of tuples returned by the `PLookupName` function in the `numGotten` parameter.

`whichOne` The sequence number of the tuple that you want the function to return. This parameter can be any integer in the range 1 through `numInBuf`.

`abEntity` A pointer to a buffer that you provide to hold the name returned by the function. This buffer must be at least 102 bytes long.

`address` A pointer to a buffer that you provide to hold the address returned by the function. The buffer must be at least 4 bytes long.

DESCRIPTION

The `NBPExtract` function extracts a name and address pair from the list of tuples that the `PLookupName` function returns. The `PLookupName` function returns the names of network-visible entities in a packed format that you cannot read from Pascal. Use the `NBPExtract` function in a loop that varies the value of the `whichOne` parameter from 1 to the total number of tuples in the list to extract all the names in the list.

ASSEMBLY-LANGUAGE INFORMATION

The `NBPExtract` function is implemented entirely in the MPW interface files. There is no assembly-language equivalent to this procedure.

RESULT CODES

`noErr`	0	No error
`extractErr`	–3104	Can't find tuple in buffer

SEE ALSO

To look up the name and address of an entity registered with NBP, use the `PLookupName` function, described on page 3-30.

For a description of the `EntityName` data type, see "Entity Name Record" on page 3-21.

For a description of the `AddrBlock` data type, see "Address Block Record" on page 3-20.

PConfirmName

The `PConfirmName` function confirms that a network-visible entity whose name you know is still available on the network and that the address associated with the name has not been changed.

```
FUNCTION PConfirmName (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioRefNum`	`Integer`	The .MPP driver reference number.
→	`csCode`	`Integer`	Always `confirmName`.
→	`interval`	`Byte`	The retry interval.
↔	`count`	`Byte`	The retry count.
→	`entityPtr`	`Ptr`	A pointer to an entity name.
→	`confirmAddr`	`AddrBlock`	The entity address.
←	`newSocket`	`Byte`	The current socket number.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .MPP driver calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .MPP driver ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum` The .MPP driver reference number. The MPW interface fills in this field.

`csCode` The command code of the .MPP command to be executed. The MPW interface fills in this field.

`interval` The retry interval to be used by NBP when it looks on the internet for the entity. The retry interval value specifies how long the function is to wait between retries in 8-tick units. A value of 7 for the `interval` field is usually sufficient ($7 \times 8 = 56$ ticks equals approximately 1 second).

`count`	The retry count to be used by NBP when it looks on the internet for the entity. The value of `count` specifies the number of times the `PConfirmName` function is to retry the operation. A retry count of 3 or 4 is usually sufficient. The `PConfirmName` function decrements this field each time it looks for names.
`entityPtr`	A pointer to an entity name that you want to confirm. The entity name must be in the format that Figure 3-5 on page 3-12 shows. You can use the `NBPSetEntity` procedure to create the entity name record.
`confirmAddr`	The last known address of the network-visible entity whose existence you wish to confirm.
`newSocket`	The current socket number of the entity. If the socket number of the entity has changed, the `PConfirmName` function returns the new socket number in this field and returns the `nbpConfDiff` result code.

DESCRIPTION

If you already know the name and address of a network-visible entity, but want to confirm that the name is still registered with NBP and that the address hasn't changed before you attempt to send data to it, you can use the `PConfirmName` function. If the address is no longer associated with the name, `PConfirmName` returns a result code of `nbpNoConfirm`, indicating that the name may have been removed from the socket. If the name is assigned to another socket, `PConfirmName` returns the current socket number in the parameter block's `newSocket` field and a result code of `nbpConfDiff`. This function generates less network traffic than the `PLookupName` function.

SPECIAL CONSIDERATIONS

Memory used for the buffer containing the entity name and the record containing the entity address belongs to `PConfirmName` until the function completes execution.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PConfirmName` function from assembly language, call the `_Control` trap macro with a value of `confirmName` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. To execute this function from assembly language, you must also specify the driver reference number.

RESULT CODES

`noErr`	0	No error
`nbpNoConfirm`	–1025	Name not confirmed
`nbpConfDiff`	–1026	Name confirmed for different socket
`tooManyReqs`	–1097	Too many concurrent requests; wait a few minutes, then try the request again
`reqAborted`	–1105	Request canceled

SEE ALSO

For a description of the `AddrBlock` data type, see "Address Block Record" on page 3-20.

To find the address of a network-visible entity whose name or address you do not already know, use the `PLookupName` function, described on page 3-30.

To cancel a name confirmation request, use the `PKillNBP` function, described on page 3-38.

PRemoveName

The `PRemoveName` function removes a previously registered name from the NBP names table.

```
FUNCTION PRemoveName (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioRefNum`	`Integer`	The .MPP driver reference number.
→	`csCode`	`Integer`	Always `removeName`.
→	`entityPtr`	`Ptr`	A pointer to an entity name.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .MPP driver calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .MPP driver ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum` The .MPP driver reference number. The MPW interface fills in this field.

`csCode` The command code of the .MPP command to be executed. The MPW interface fills in this field.

`entityPtr` A pointer to the name of the network-visible entity that you wish to remove from the names table. The name must be in the format shown in Figure 3-5 on page 3-12. You cannot use any wildcard characters in the name.

DESCRIPTION

When you close a socket or terminate an application or process that you registered in the NBP names table as a network-visible entity, you must use the `PRemoveName` function to remove the name from the names table.

To remove the names table entry, you assign to the `entityPtr` field of the parameter block a pointer to a fully qualified entity name. The entity name is a packed array of Pascal strings. Because the object, type, and zone names in this format are of arbitrary length, you cannot create this record in Pascal (which requires you to declare the length of character strings when you define the record). You can use the `NBPSetEntity` procedure to create this record, or you can provide `PRemoveName` with a pointer to the names table entry record that you used to register the name.

SPECIAL CONSIDERATIONS

Memory used for the buffer containing the entity name belongs to the `PRemoveName` function until the function completes execution and must be nonrelocatable. After you remove the names table entry, you can reuse the memory or release it.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PRemoveName` function from assembly language, call the `_Control` trap macro with a value of `removeName` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. To execute this function from assembly language, you must also specify the driver reference number.

RESULT CODES

`noErr`	0	No error
`nbpNotFound`	–1028	Name not found
`reqAborted`	–1105	Request canceled

SEE ALSO

To create an entity name record of the form required by the `PRemoveName` function, use the `NBPSetEntity` procedure, described on page 3-28.

PKillNBP

The `PKillNBP` function cancels NBP function calls to the `PLookupName`, `PRegisterName`, or `PConfirmName` function.

```
FUNCTION PKillNBP (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioRefNum`	`Integer`	The .MPP driver reference number.
→	`csCode`	`Integer`	Always `killNBP`.
→	`nKillQEl`	`Ptr`	A pointer to a queue element.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .MPP driver calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .MPP driver ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum` The .MPP driver reference number. The MPW interface fills in this field.

`csCode` The command code of the .MPP command to be executed. The MPW interface fills in this field.

`nKillQEl` A pointer to the MPP parameter block for the NBP request you want to cancel.

DESCRIPTION

When you call the `PLookupName`, `PRegisterName`, or `PConfirmName` function asynchronously, the Device Manager puts your request in the .MPP driver's queue with other requests. If you want to cancel a pending NBP request, you pass a pointer to the parameter block for that request to the `PKillNBP` function.

If the function's parameter block is in the .MPP driver's queue waiting for the function to be executed, the `PKillNBP` function deletes the entry from the queue and returns a function result of `noErr`. The function whose parameter block is deleted completes execution and returns a function result of `reqAborted`, indicating that the function was canceled.

If the function has already been executed, that is, it is no longer in the queue, `PKillNBP` returns a function result of `cbNotFound`, indicating that the parameter block for the function to be canceled was not in the .MPP driver's queue.

The function also calls the completion routine for the canceled request with the result code `reqAborted` (–1105) in the D0 register.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PKillNBP` function from assembly language, call the `_Control` trap macro with a value of `killNBP` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. To execute this function from assembly language, you must also specify the driver reference number.

RESULT CODES

`noErr`	0	No error
`cbNotFound`	–1102	NBP queue element not found

Summary of NBP

Pascal Summary

Constants

```
CONST
   {.MPP driver unit and reference number}
   mppUnitNum     =     9;          {MPP driver unit number}
   mppRefNum      =   -10;          {MPP reference number}

   {csCodes for NBP}
   confirmName    =   250;          {confirm name}
   lookupName     =   251;          {lookup name}
   removeName     =   252;          {remove name from names table}
   registerName   =   253;          {register name in names table}
   killNBP        =   254;          {kill outstanding NBP request}
```

Data Types

Address Block Record

```
   AddrBlock =
   PACKED RECORD
      aNet:       Integer;          {network number}
      aNode:      Byte;             {node ID}
      aSocket:    Byte;             {socket number}
   END;
```

Names Table Entry Record

```
TYPE  NamesTableEntry =
   RECORD
      qLink:      QElemPtr;         {pointer to next NTE in names table}
      nteAddress: AddrBlock;        {pointer to this names table entry}
      nteData: PACKED ARRAY[1..100] OF Char;
                                    {names table entry}
   END;
```

Entity Name Record

```
EntityName =
RECORD
   objStr:        Str32;      {object name}
   typeStr:       Str32;      {type name}
   zoneStr:       Str32;      {zone name}
END;
EntityPtr = ^EntityName;
```

MPP Parameter Block for NBP

```
MPPParmType    =     (...RegisterNameParm, LookupNameParm,
                        ConfirmNameParm,RemoveNameParm...);
TYPE MPPParamBlock =
   PACKED RECORD
      qLink:               QElemPtr;      {reserved}
      qType:               Integer;       {reserved}
      ioTrap:              Integer;       {reserved}
      ioCmdAddr:           Ptr;           {reserved}
      ioCompletion:        ProcPtr;       {completion routine}
      ioResult:            OSErr;         {result code}
      ioNamePtr:           StringPtr;     {reserved}
      ioVRefNum:           Integer;       {reserved}
      ioRefNum:            Integer;       {driver reference number}
      csCode:              Integer;       {primary command code}
   CASE MPPParmType OF
      RegisterNameParm,
      LookupNameParm,
      ConfirmNameParm,
      RemoveNameParm:
         (interval:        Byte;          {retry interval}
         count:            Byte;          {retry count}
         entityPtr:        Ptr;           {pointer to entity name or }
                                          { names table entry}
         CASE MPPParmType OF
            RegisterNameParm:
               (verifyFlag:   Byte;       {verify uniqueness of name or not}
               filler3:       Byte;)
            LookupNameParm:
               (retBuffPtr:   Ptr;        {pointer to return buffer}
               retBuffSize:   Integer;    {return buffer size}
               maxToGet:      Integer;    {matches to get}
               numGotten:     Integer;)   {matches gotten}
```

```
            ConfirmNameParm:
               (confirmAddr:  AddrBlock;  {pointer to entity name}
               newSocket:     Byte;       {socket number}
               filler4:       Byte);
            )
      KillNBPParm:
         (nKillQEl:        Ptr;)       {pointer to queue element to cancel}
   END;

MPPPBPtr    =     ^MPPParamBlock;
```

Routines

Registering an Entity

```
PROCEDURE NBPSetNTE          (ntePtr: Ptr; nbpObject,nbpType,nbpZone: Str32;
                              socket: Integer);
FUNCTION PRegisterName       (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

Handling Name and Address Requests

```
PROCEDURE NBPSetEntity       (buffer: Ptr; nbpObject,nbpType,nbpZone: Str32);
FUNCTION PLookupName         (thePBptr: MPPPBPtr; async: Boolean): OSErr;
FUNCTION NBPExtract          (theBuffer: Ptr; numInBuf: Integer; whichOne:
                              Integer; VAR abEntity: EntityName; VAR address:
                              AddrBlock): OSErr;
FUNCTION PConfirmName        (thePBptr: MPPPBPtr; async: Boolean): OSErr;
FUNCTION PRemoveName         (thePBptr: MPPPBPtr; async: Boolean): OSErr;
FUNCTION PKillNBP            (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

C Summary

Constants

```
/*NBP parameter constants*/
#define MPPioCompletion MPP.ioCompletion
#define MPPioResult MPP.ioResult
#define MPPioRefNum MPP.ioRefNum
#define MPPcsCode MPP.csCode
#define NBPinterval NBP.interval
#define NBPcount NBP.count
```

```
#define NBPntQElPtr NBP.NBPPtrs.ntQElPtr
#define NBPentityPtr NBP.NBPPtrs.entityPtr
#define NBPverifyFlag NBP.parm.verifyFlag
#define NBPretBuffPtr NBP.parm.Lookup.retBuffPtr
#define NBPretBuffSize NBP.parm.Lookup.retBuffSize
#define NBPmaxToGet NBP.parm.Lookup.maxToGet
#define NBPnumGotten NBP.parm.Lookup.numGotten
#define NBPconfirmAddr NBP.parm.Confirm.confirmAddr
#define NBPnKillQEl NBPKILL.nKillQEl
#define NBPnewSocket NBP.parm.Confirm.newSocket

enum {                                  /*.MPP driver unit and reference */
                                        /* number*/
   mppUnitNum     =     9,              /*.MPP driver unit number*/
   mppRefNum      =     -10};           /*MPP reference number*/

enum {                                  /*.MPP csCodes*/
   confirmName    =     250,            /*confirm name*/
   lookupName     =     251,            /*lookup name*/
   removeName     =     252,            /*remove name from names table*/
   registerName   =     253,            /*register name in names table*/
   killNBP        =     254};           /*kill outstanding NBP request*/
```

Data Types

Address Block Record

```
struct AddrBlock {
   short             aNet;       /*network name*/
   unsigned char     aNode;      /*node name*/
   unsigned char     aSocket;    /*socket number*/
};

typedef struct AddrBlock AddrBlock;
```

Names Table Entry Data Structure

```
struct {
   Ptr            qNext;           /*pointer to next names table element*/
   NTElement      nt;
}NamesTableEntry;
```

Entity Name Record

```
struct EntityName {
   Str32          objStr;        /*object name*/
   char           pad1;          /*Str32's aligned on even word boundaries*/
   Str32          typeStr;       /*type name*/
   char           pad2;
   Str32          zoneStr;       /*zone name*/
   char           pad3;
};

typedef struct EntityName EntityName;
typedef EntityName *EntityPtr;
```

MPP Parameter Block for NBP

```
#define MPPATPHeader \
   QElem          *qLink;        /*reserved*/\
   short          qType;         /*reserved*/\
   short          ioTrap;        /*reserved*/\
   Ptr            ioCmdAddr;     /*reserved*/\
   ProcPtr        ioCompletion;  /*completion routine*/\
   OSErr          ioResult;      /*result code*/\
   long           userData;      /*command result (ATP user bytes)*/\
   short          reqTID;        /*request transaction ID*/\
   short          ioRefNum;      /*driver reference number*/\
   short          csCode;        /*primary command code*/

typedef struct {
   MPPATPHeader
}MPPparms;

typedef struct {
   MPPATPHeader
   char           interval;         /*retry interval*/
   char           count;            /*retry count*/
   union {
      Ptr            ntQElPtr;      /*pointer to queue element to cancel*/
      Ptr            entityPtr;
                                    /*pointer to entity name or names */
                                    /* table entry*/

      } NBPPtrs;
```

```
   union {
      char           verifyFlag;    /*verify uniqueness of name or not*/
   struct {
      Ptr            retBuffPtr;    /*pointer to return buffer*/
      short          retBuffSize;   /*return buffer size*/
      short          maxToGet;      /*matches to get*/
      short          numGotten;     /*matches gotten*/
   } Lookup;
   struct {
         AddrBlock      confirmAddr; /*pointer to entity name*/
         char           newSocket;   /*socket number*/
      } Confirm;
   } parm;
}NBPparms;


struct {
   MPPATPHeader
   Ptr            nKillQEl;
                                  /*pointer to queue element to cancel*/
}NBPKillparms;

union ParamBlockRec {
   MPPparms       MPP;            /*general MPP parms*/
   NBPparms       NBP;            /*NBP calls*/
   NBPKillparms   NBPKILL;        /*cancel call to NBP*/
};
typedef MPPParamBlock *MPPPBPtr;
```

Routines

Registering an Entity

```
pascal void NBPSetNTE       (Ptr ntePtr, Ptr nbpObject, Ptr nbpType,
                             Ptr nbpZone, short socket);
pascal OSErr PRegisterName  (MPPPBPtr thePBpt, Boolean async);
```

Handling Name and Address Requests

```
pascal void NBPSetEntity    (Ptr buffer, Ptr nbpObject, Ptr nbpType,
                             Ptr nbpZone);
pascal OSErr PLookupName    (MPPPBPtr thePBptr, Boolean async);
```

```
pascal OSErr NBPExtract     (Ptr theBuffer, short numInBuf, short whichOne,
                             EntityName *abEntity, AddrBlock *address);
pascal OSErr PConfirmName   (MPPPBPtr thePBptr, Boolean async);
pascal OSErr PRemoveName    (MPPPBPtr thePBptr, Boolean async);
pascal OSErr PKillNBP       (MPPPBPtr thePBptr, Boolean async);
```

Assembly-Language Summary

Constants

Unit Number for the .MPP Driver

```
mppUnitNum        EQU   9           ;MPP unit number
```

NBP Symbolic Characters

```
equals            EQU   '='         ;wildcard symbol
NBPWildCard       EQU   '≈'         ;wildcard symbol
star              EQU   '*'         ;"This zone" symbol
```

NBP Command Codes

```
registerName      EQU   253         ;register name in names table
lookupReply       EQU   242         ;used internally
lookupName        EQU   251         ;look up an NBP name
confirmName       EQU   250         ;confirm name
removeName        EQU   252         ;remove name from names table
killNBP           EQU   254         ;kill outstanding NBP request
```

NBP Packet

```
nbp               EQU   $02         ;DDP protocol type code for NBP
nbpControl        EQU   0           ;control code
nbpTCount         EQU   0           ;tuple count
nbpID             EQU   1           ;NBP ID
nbpTuple          EQU   2           ;start of the first tuple
```

NBP Tuple Header Offsets

```
tupleNet          EQU   0           ;offset to network number (word)
tupleNode         EQU   2           ;offset to node ID (byte)
tupleSkt          EQU   3           ;offset to socket number (byte)
```

```
tupleEnum         EQU   4           ;offset to enumerator (byte)
tupleName         EQU   5           ;offset to name part of tuple (byte)
tupleAddrSz       EQU   5           ;tuple address field size
```

NBP Packet Types

```
brRq              EQU   1           ;broadcast request
lkUp              EQU   2           ;lookup request
lkUpReply         EQU   3           ;lookup reply
```

NBP Names Information Socket (NIS) Number

```
nis               EQU   2           ;NIS number
```

Maximum Number of Tuples in NBP Packet, Maximum Size of a Tuple Name

```
tupleMax          EQU   15          ;maximum number of tuples returned from
                                    ; a lookup request
NBPMaxTupleSize   EQU   32          ;maximum size of a tuple name
```

Data Structures

MPP Parameter Block Common Fields for NBP

0	qLink	long	reserved
4	qType	word	reserved
6	ioTrap	word	reserved
8	ioCmdAddr	long	reserved
12	ioCompletion	long	address of completion routine
16	ioResult	word	result code
18	ioNamePtr	long	reserved
22	ioVRefNum	word	reserved
24	ioRefNum	word	driver reference number

PRegisterName Parameter Variant

26	csCode	word	command code; always `registerName`
28	interval	byte	retry interval
29	count	byte	retry count
30	entityPtr (ntQElPtr)	long	names table queue element pointer
34	verifyFlag	byte	verify name flag
40	filler	byte	reserved

PLookupName Parameter Variant

26	`csCode`	word	command code; always `lookupName`
28	`interval`	byte	retry interval
29	`count`	byte	retry count
30	`entityPtr`	long	pointer to entity name
34	`retBuffPtr`	long	pointer to return data buffer
38	`retBuffSize`	word	size in bytes of return buffer
40	`maxToGet`	word	maximum number of matches to get
42	`numGotten`	word	number of matches returned

PConfirmName Parameter Variant

26	`csCode`	word	command code; always `confirmName`
28	`interval`	byte	retry interval
29	`count`	byte	retry `count`
30	`entityPtr`	long	pointer to entity name
34	`confirmAddr`	long	address of names table entry to confirm
38	`newSocket`	byte	socket number, if different from specified one
39	`filler`	byte	reserved

PRemoveName Parameter Variant

26	`csCode`	word	command code; always `removeName`
28	`filler`	word	reserved
30	`entityPtr`	long	pointer to entity name

PKillNBP Parameter Variant

26	`csCode`	word	command code; always `killNBP`
28	`nKillQEl`	long	pointer to queue element to remove

Result Codes

`noErr`	0	No error
`nbpNoConfirm`	–1025	Name not confirmed
`nbpConfDiff`	–1026	Name confirmed for different socket
`nbpDuplicate`	–1027	Name already exists
`nbpNotFound`	–1028	Name not found
`tooManyReqs`	–1097	Too many concurrent requests; wait a few minutes, then try the request again
`cbNotFound`	–1102	NBP queue element not found
`reqAborted`	–1105	Request canceled
`extractErr`	–3104	Can't find tuple in buffer