# Network Services Location Manager Developer's Kit

Technical Publications

 Apple Computer, Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD “AS IS,” AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

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

# Contents

# Figures, Listings, and Tables

# About This Manual

This document describes the programming interface for the Network Services Location (NSL) Manager. The NSL Manager provides a protocol-independent way for applications to discover available network services with minimal network traffic.

## Conventions Used in This Manual

The Courier font is used to indicate function names, code, and text that you type. This manual includes special text elements to highlight important or supplemental information:

**Note**

Text set off in this manner presents sidelights or interesting points of information. ◆

**IMPORTANT**

Text set off in this manner—with the word Important—presents important information or instructions. ▲

▲ **WARNING**

Text set off in this manner—with the word Warning—indicates potentially serious problems. ▲

## For more information

The following sources provide additional information that is important for NSL developers:

- *Inside Macintosh* , available online at http://devworld.apple.com/techinfo/techdocs/mac/mac.html
- *NSL Network Administrators Guide* , which tells administrators how to configure DNS and SLP servers so they can participate in NSL lookups (available with the final version of the NSL SDK)
- *DNS and Bind* by Paul Albitz and Cricket Liu, O'Reilly & Associates, Inc. 1994
- RFC 2165, Service Location Protocol, available at http://www.isi.edu/rfc-editor/rfc.html

CHAPTER 1

# Network Services Location Manager

## Contents

CHAPTER 1

# Network Services Location Manager

The Network Services Location (NSL) Manager provides a protocol-independent way for applications to discover available network services with minimal network traffic.

The NSL Manager provides

- AppleTalk-like ease-of-use for the dynamic discovery of traditional and non-traditional network services
- Support for accepted and proposed industry standards, including Domain Name Service (DNS) and Service Location Protocol (SLP)
- A flexible, expandable architecture that can be easily leveraged by client and server applications

A wide variety of applications will become easier to use when they call the NSL Manager. For example,

- Instead of requiring the user to type a URL to locate a web server, a browser application that calls the NSL Manager could have an “Open Location” command that polls the network for Hypertext Transfer Protocol (HTTP) servers and displays a list of HTTP universal resource locators (URLs) from which the user can select a particular URL.
- Collaboration software, such as a video-conferencing server, would register itself as an available service on the corporate Intranet. The users of client video-conferencing software could then search the Intranet for available conferences and join a particular conference without having to remember a cryptic URL or Internet Protocol (IP) address.

The NSL Manager acts as an intermediary between the providers of network services and applications that want information about such services. It also registers network services that make registration requests.

This chapter describes how you can use the NSL Manager to

- add network-service search functionality to your application
- register a network service with the NSL Manager so that it can be found in searches

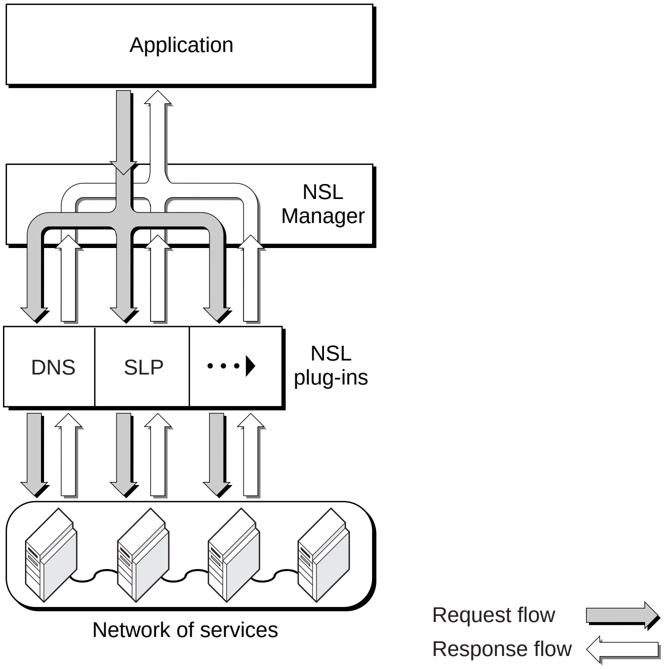
The NSL Manager will be available on all PowerPC-based computers that run a version of Mac OS 8.5 or later and will be accompanied by two NSL plug-ins that perform searches: DNS and SLP. Additional NSL plug-ins may also become available from Apple Computer or third-party developers.

The section “About the NSL Manager” explains the relationship between applications that call the NSL Manager and the NSL plug-ins. The section “About NSL Plug-ins” describes the NSL plug-ins that come with the NSL SDK. The section “Searching for Network Services” provides sample code of an application calling the NSL Manager to search for services.

## About the NSL Manager

The NSL Manager provides a protocol-independent interface for applications that need to locate network services and plug-ins that search for network services. Figure 1-1 illustrates the relationship between applications, the NSL Manager, and the NSL plug-ins.

**Figure 1-1** Overview of a network service lookup



Applications that search for services can focus the search by specifying two values:

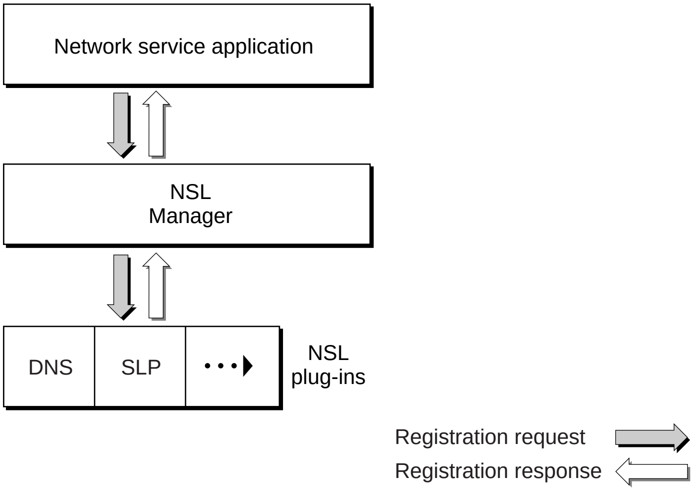
- a services list, which is an NSL data type that allows the application to specify the services that are to be searched for.
- a neighborhood, which is an NSL data type that allows the application to define the scope of the search. For example, a neighborhood data type may contain a domain name, such as apple.com.

The following steps outline the flow of a service lookup:

1. The application creates a lookup request and calls the NSL Manager's `NSLStartServicesLookup` function.
2. The NSL Manager receives the request and passes it to those NSL plug-ins that are capable of responding to the request.
3. Each NSL plug-in that receives the request starts to look for the specified services.
4. Providers of services send their responses to the NSL plug-ins.
5. The NSL plug-ins pass the responses to the NSL Manager.
6. The NSL Manager passes the responses to the application that initiated the lookup. If more than one plug-in responds, the NSL Manager returns the responses to the application in a single response stream.

Applications that provide services can register themselves with the NSL Manager as shown in Figure 1-2.

**Figure 1-2** Overview of a service registration



The following steps outline the flow of a service registration:

1. The application creates a service registration request and calls the NSL Manager's `NSLRegisterService` function.
2. The NSL Manager receives the request and passes it to the NSL plug-in that is capable of registering the service.
3. The NSL plug-in receives the request and registers the service.
4. The NSL Manager returns a value to the application indicating that the service was registered successfully.

**Note**
Currently, the SLP plug-in is the only plug-in provided by Apple Computer that can register services. ◆

# About NSL Plug-ins

An NSL plug-in is an extension that searches for services. It makes itself available to the NSL Manager when the NSL Manager is initialized, and it resides in memory only when it is responding to lookup requests from applications.

**Note**
The Extensions Manager can be used to enable and disable individual NSL plug-ins. ◆

The NSL Manager can pass lookup requests to any plug-in that adheres to the NSL Manager API.

When the NSL Manager is initialized, each NSL plug-in provides the following information to the NSL Manager:

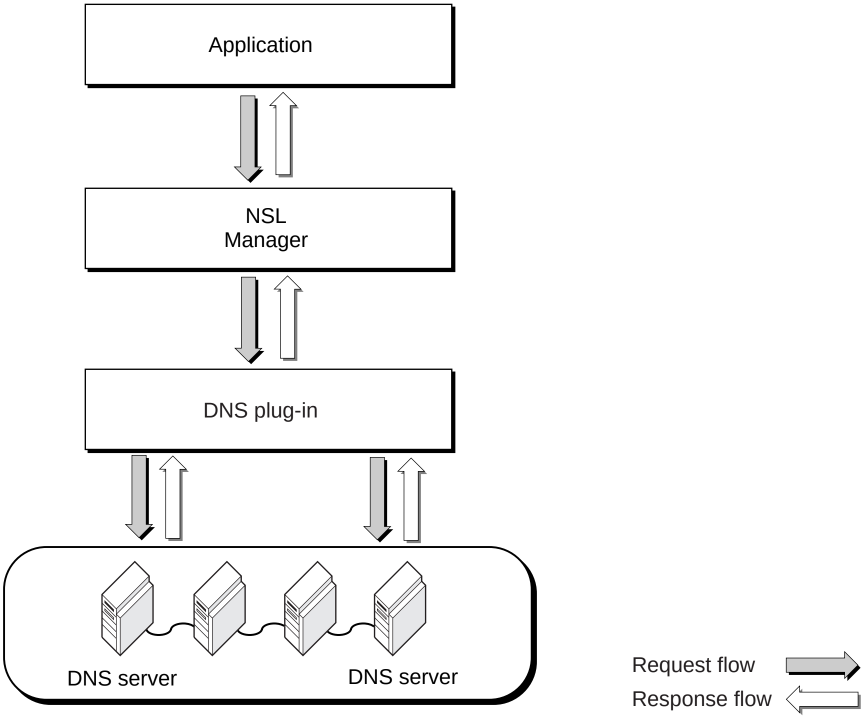
- the types of services the plug-in can search for, such as HTTP
- the protocol the plug-in uses to conduct searches, such as DNS

The NSL SDK comes with two NSL plug-ins: DNS and SLP.

## About the DNS Plug-in

The DNS plug-in allows applications to receive lists of services from DNS servers. The information about each service is taken from the “well-known services” field of each DNS entry. Figure 1-3 shows the flow of a DNS lookup.

**Figure 1-3** Flow of a DNS lookup



The DNS plug-in provides the following routines for the NSL Manager to call:

- An initialization routine that allocates memory and opens network connections to DNS servers
- A deinitialization routine that deallocates memory and closes network connections
- A start-neighborhood-lookup routine that starts a neighborhood lookup
- A start-services-lookup routine that starts a service lookup
- A continue-lookup routine that resumes a lookup for services or neighborhoods that has paused in order to deliver lookup results to the application
- A cancel-lookup routine that cancels an ongoing lookup

- An error-number conversion routine that provides a pair of strings describing the error and a possible solution for any error number that the plug-in may return
- An information routine that provides details about the services and protocols the plug-in supports, as well as a comment string that describes the services and protocol the plug-in supports

## About the SLP Plug-in

The SLP plug-in uses the Service Location Protocol to locate services. The Service Location Protocol is an emerging Internet Engineering Task Force (IETF) protocol designed to simplify the discovery and use of network resources. SLP is well-suited for client-server applications and for establishing connections between network peers that offer or consume generic services. SLP supports servers that register services dynamically as well as clients that use multicast protocols to broadcast for services.

The SLP plug-in accepts service registrations from applications that provide network services running on the local host. When the SLP plug-in registers a service, it creates an SLP Service Agent for the service. The Service Agents listen for lookup requests and respond appropriately when the SLP plug-in queries them.

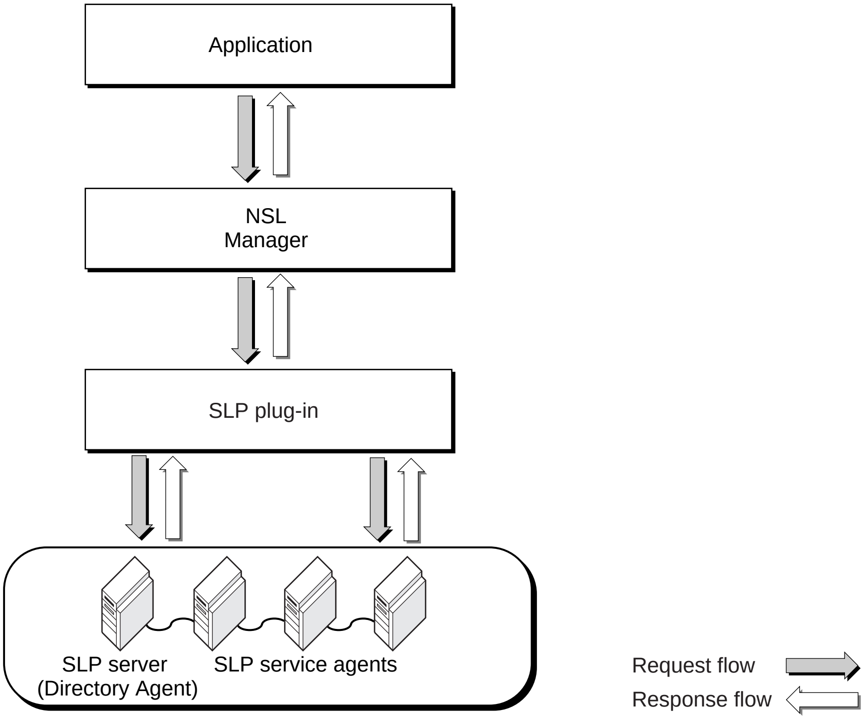
The SLP plug-in also listens for and registers with any SLP Directory Agent Servers (DAs) that may be present on the local subnet. The SLP plug-in then listens for and registers with any other DAs that may announce their availability on the local subnet.

**Note**

When the SLP plug-in is first loaded into memory, it uses IP multicast to locate DAs. Any routers on the local subnet must be configured to support IP multicast. ◆

If a network has a DA, Service Agents register themselves with the DA. The SLP plug-in can then query the DA directly, thereby minimizing network traffic. In Figure 1-4, the SLP plug-in can bypass the Service Agents and query the DA directly. If the DA becomes unavailable, the SLP plug-in will query each Service Agent individually.

**Figure 1-4** Flow of an SLP lookup



Like the DNS plug-in, the SLP plug-in provides routines that initialize and deinitialize the plug-in, start, continue, and cancel a service or neighborhood lookup, return a pair of strings that describe an error condition and a possible solution for any error code that the SLP plug-in may return, and a routine that returns information that describes the plug-in's capabilities. The SLP plug-in also provides routines to register and deregister services.

For more information about SLP, see RFC 2165.

# Searching for Network Services

To search for network services, an application calls NSLOpenNavigationAPI to initialize the NSL Manager, as shown in Listing 1-1.

**Listing 1-1** Initializing the NSL Manager

```
OSStatus status;
status = NSLOpenNavigationAPI( &gOurClientRef );
```

The NSL Manager returns a client reference that the application uses to prepare a lookup request and to call NSLCloseNavigationAPI when the application no longer needs to make lookup requests.

Next, the application calls NSLMakeNewServicesList to create a services list and calls NSLMakeRequestPB to convert the resulting services list into a request parameter block, as shown in Listing 1-2.

**Listing 1-2** Creating a request parameter block

```
NSLServicesList serviceList =       NULL;
serviceList = NSLMakeNewServicesList( "http,ftp" );
iErr.theErr = NSLMakeRequestPB( serviceList, "", &newDataPtr );
```

In Listing 1-2, the application creates a services list that specifies that HTTP and FTP services are to be searched for. If the application doesn’t specify any services, all services will be searched for. The application then calls NSLMakeRequestPB with the services list as a parameter. The NSLMakeRequestPB function formats the services list in a way that allows any plug-in to parse the services list properly.

Next, the application creates a lookup request by calling NSLPrepareRequest, as shown in Listing 1-3.

**Listing 1-3** Preparing an NSL lookup request

```
long bufLen =                4096;
char* buffer =               NULL;
NSLRequestRef                ourRequestRef;
ClientAsyncInfoPtr           ourAsyncInfo;
NSLError iErr =              kNSLErrorNoErr;

buffer = NewPtr( bufLen );

iErr = NSLPrepareRequest( NULL, NULL, gOurClientRef, &ourRequestRef,
                               buffer, bufLen, &ourAsyncInfo );
if ( iErr.theErr )
{
    // Handle error.
}
```

Calling `NSLPrepareRequest` returns a `requestRef` and sets up a `ClientAsyncInfo` structure for this request. The application uses the `ClientAsyncInfo` structure to search for neighborhoods and services. The application can control the way the search is conducted by specifying

- a maximum time for the search
- an alert threshold (that is, return search results whenever a certain number if items have been returned)
- an alert interval (that is, return search results whenever a specified time elapses)

The NSL Manager uses the `ClientAsyncInfo` structure to convey search results and status information about the search from the plug-in to the application.

In Listing 1-4, the application calls `NSLStartNeighborhoodLookup` to obtain the first available neighborhood on the Intranet and calls `NSLContinueLookup` until it has obtained all of the available neighborhoods on the Intranet.

**Listing 1-4** Searching for neighborhoods

```
// Set the values of the ourAsyncInfo parameter block
ourAsyncInfo->clientContextPtr = NULL;
ourAsyncInfo->maxSearchTime = 0;// no max search time
```

```
ourAsyncInfo->alertInterval = 0; // no alert interval
ourAsyncInfo->alertThreshold = 1;// return after each item

if ( iErr.theErr == noErr )
    iErr = NSLStartNeighborhoodLookup( ourRequestRef, neighborhood,
               ourAsyncInfo );
    do {
    if ( iErr.theErr == noErr && ourAsyncInfo->totalItems > 0 )
        {
        while ( NSLGetNextNeighborhood( ourAsyncInfo, &nhPtr,
                    &nhLength ) )
        {
                if ( nhLength > 0 && nhLength < kBufferLength  )
                {
                p2cstr( (unsigned char*) nhPtr );
                    // Each neighborhood is a data structure that
                    // starts with a pstring of the name
                    printf( "%s\r", nhPtr );
            }
            else
            {
                done = true;
            }
        }
            if ( ourAsyncInfo->searchState == kNSLSearchStateComplete )
            done = true;
        else
            iErr = NSLContinueLookup( ourAsyncInfo );
    }

} while ( !iErr.theErr && !done );

    if ( buffer )
        DisposePtr(buffer);
}
```

The application could display the name of each neighborhood and allow the user to select one.

In Listing 1-5, the application calls `NSLStartServicesLookup` to start the service lookup in the selected neighborhood, as specified by the `neighborhood` parameter. The ourRequest parameter was created earlier by calling NSLPrepareRequest and the newDataPtr parameter was created earlier by calling NSLMakeRequestPB.

The application continues to call NSLContinueLookup until it has received information about all of the services that match the search criteria.

**Listing 1-5** Searching for services

```
iErr = NSLStartServicesLookup( ourRequestRef, neighborhood, newDataPtr,
       ourAsyncInfo );
do {
    if ( iErr.theErr == noErr && ourAsyncInfo->totalItems > 0 )
    {

    while ( NSLGetNextUrl( ourAsyncInfo, &urlPtr, &urlLength ) )
    {
        if ( urlLength > 0 )
                {
                    // Process resultBuffer.
                }
                else
{
                    done = true;
                }
            }
            if ( ourAsyncInfo->searchState == kNSLSearchStateComplete )
                done = true;
            else
                iErr = NSLContinueLookup( ourAsyncInfo );
        }
    } while ( !iErr.theErr && !done );
```

When the lookup is complete, the application reclaims memory allocated for the services list, the request parameter block, and the lookup request, as shown in Listing 1-6.

**Listing 1-6** Reclaiming memory

```
NSLDisposeServicesList(serviceList);
NSLDeleteRequest(ourRequestRef);
NSLFreeTypedDataPtr(newDataPtr);
```

When the application has no need to make additional lookups, it calls `NSLCloseNavigationAPI` to close the NSL Manager, as shown in Listing 1-7.

**Listing 1-7** Deinitializing the NSL Manager

```
NSLCloseNavigationAPI(gOurClientRef);
```

If this application is the last application that has a requirement for a particular plug-in, the NSL Manager unloads that plug-in from memory.

# Network Services Location Manager Reference

## Contents

# NSL Manager Constants and Data Types

## ClientAsyncInfo Structure

The ClientAsyncInfo structure contains information about how a neighborhood or a service lookup is to be conducted and where lookup results are to be stored. You obtain a pointer to a ClientAsyncInfo structure by calling NSLPrepareRequest (page 2-35), and you pass that pointer as a parameter when you call NSLStartNeighborhoodLookup (page 2-37), NSLStartServicesLookup (page 2-40), or NSLContinueLookup (page 2-42).

Before you call NSLStartServicesLookup or NSLStartNeighborhoodLookup, you can modify the way in which the lookup is conducted by changing certain values in the ClientAsyncInfo structure. However, once you call NSLStartServicesLookup or NSLStartNeighborhoodLookup, you should not modify the ClientAsyncInfo structure.

When NSLStartServicesLookup, NSLStartServicesLookup, or NSLContinueLookup returns, or when your application's notification routine is called, the ClientAsyncInfo structure contains information about the status of the lookup and any search results.

```
typedef struct ClientAsyncInfo
{
    void*           clientContextPtr;
    void*           mgrContextPtr;
    char*           resultBuffer;
    long            bufferLen;
    long            maxBufferSize;
    UInt32          startTime;
    UInt32          intStartTime;
    UInt32          maxSearchTime;
    UInt32          alertInterval;
    UInt32          totalItems;
    UInt32          alertThreshold;
    NSLSearchState  searchState;
    NSLError        searchResult;
    UInt32          searchDataType
} ClientAsyncInfo, *ClientAsyncInfoPtr;
```

**Field descriptions**

`clientContextPtr` A value set by the application for its own use.

`mgrContextPtr` A value set by the NSL Manager for its own use.

`resultBuffer` A pointer to the buffer that contains lookup results.

`bufferLen` The number of bytes in `resultBuffer` that contain valid data.

`maxBufferSize` The length of `resultBuffer`.

`startTime` Used by the NSL Manager for internal purposes. Your application should not modify this field.

`intStartTime` Used by the NSL Manager for internal purposes. Your application should not modify this field.

`maxSearchTime` An application-specified limit in ticks on the total amount of time that is to be expended on the search. The default value is zero, which indicates that the search time is not to be limited. The value of `maxSearchTime` does not override any limit that a plug-in may impose.

`alertInterval` An application-specified value that defines in ticks the interval at which the application's notification routine is to be called or the interval at which `NSLStartServicesLookup`, `NSLStartNeighborhoodLookup`, or `NSLContinueLookup` are to return. The default value is zero, which indicates that no alert interval is specified.

`totalItems` The total number of items in `resultbuffer`.

`alertThreshold` An application-specified value that causes the application's notification routine to be called or `NSLStartServicesLookup`, `NSLStartNeighborhoodLookup`, or `NSLContinueLookup` to return whenever the specified number of items have been placed in `resultBuffer`. Typically, applications that cause `NSLStartServicesLookup` or `NSLStartNeighborhoodLookup` to operate asynchronously set `alertThreshold` to 1, and applications that cause `NSLStartNeighborhood` or `NSLStartServicesLookup` to operate synchronously set `alertThreshold` to zero, which indicates that no alert threshold is specified. The default value is zero.

searchState
A value that describes the current search state. The value can be one of the following:

```
kNSLSearchStateBufferFull= 1,
kNSLSearchStateOnGoing = 2,
kNSLSearchStateComplete = 3,
kNSLSearchStateStalled = 4
```

searchResult
An `NSLError` structure containing an error code that the NSL Manager or a plug-in may have returned.

searchDataType
An event code that indicates whether the information stored in this `ClientAsyncInfo` structure pertains to a neighborhood lookup (`kNSLNeighborhoodLookupDataEvent`) or a service lookup (`kNSLServicesLookupDataEvent`).

## NSLError Structure

The `NSLError` structure is used by certain NSL Manager functions to return an error code as well as contextual information about that error code.

```
typedef struct NSLError {
    OSStatus    theErr;
    UInt32  theContext;
} NSLError *NSLErrorPtr;
```

### Field descriptions

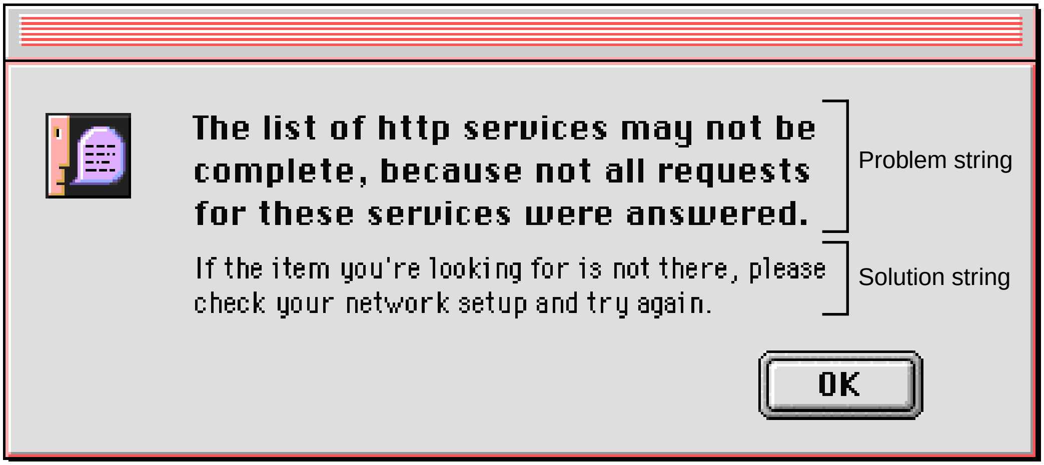
theErr
The error code.

theContext
A value used by the NSL Manager to determine whether it generated the error code or whether a plug-in generated error code. If a plug-in generated the error code, the value of `theContext` allows the NSL Manager to identify the responsible plug-in.

Comparing the constant `kNSLErrorNoErr` to the value returned by an function that returns an `NSLError` structure is a simple way to determine whether an error occurred.

If you want to display information about the error to the user, your application should call `NSLErrorToString` (page 2-44) to obtain two strings — a problem string and a solution string. To display the strings, use a movable modal dialog box, as shown in Figure 2-1.

**Figure 2-1** Standard alert dialog box



The list of http services may not be complete, because not all requests for these services were answered.

If the item you're looking for is not there, please check your network setup and try again.

Table 2-1 lists the problem and solution strings for error conditions that commonly occur.

**Table 2-1** Problem and solution strings for some NSL error conditions

| Condition | Description | Problem string | Solution string |
|---|---|---|---|
| Service not available | The application calls `NSLStartServicesLookup`, but when the plug-in tries to communicate with the servers of that service, the servers do not answer. This condition might be due to a protocol-specific configuration error that prevented the server from receiving the plug-in's query. For example, the address of a DNS server address in the TCP/IP control panel is not correct. | The list of <*type of service*> services may not be complete, because not all requests for these services were answered. | If the item you're looking for is not there, please check your network setup and try again. |
| Timeout | The application calls `NSLStartServicesLookup` and the plug-in initiates the query but does not receive any results. | The list of <*type of service*> services may not be complete because the network search timed out. | Your <*type of network*> network may have been interrupted. Please try again later. |
| Network failure | The application calls `NSLStartServicesLookup` and the plug-in detects that the network is down. | Your network is not responding. | Your <*type of network*> network may have been interrupted. Please try again later. |
| Connection failure | The application calls `NSLStartServicesLookup` but the plug-ins cannot communicate with the servers because there is no appropriate network connection. | You can not connect to your <*type of network*> network. | Check your network settings and make sure all networking cables are properly attached. Then try again. |
| Not enough memory | The application calls `NSLStartServicesLookup` but there is not enough memory to load one or more plug-ins or for one or more plug-ins to initialize itself. | The last command could not be completed because there is not enough memory. | <*standard low memory instructions*> |

# NSL Manager Functions

## Managing NSL Manager Sessions

### NSLOpenNavigationAPI

Open a session with the NSL Manager.

```
OSStatus NSLOpenNavigationAPI (NSLClientRef* newref);
```

| | |
|---|---|
| `newref` | On input, a pointer to an `NSLClientRef` in which the NSL Manager returns a value that your application uses in subsequent `NSLPrepareRequest` (page 2-35) and `NSLCloseNavigationAPI` calls (page 2-33). |
| *function result* | A value of `noErr` indicates that the session was opened and all available plug-ins loaded successfully. A value of `kNSLSomePluginsFailedToLoad` indicates that the session was opened and at least one plug-in loaded successfully. If `NSLOpenNavigationAPI` returns any of the following error codes, your application should not call any other NSL Manager functions: `kNSLNotInitialized`, `kNSLInsufficientSysVer`, `kNSLInsufficientOTVer`, and `kNSLPluginLoadFailed`. |

**DISCUSSION**

The `NSLOpenNavigationAPI` function opens a session with the NSL Manager and returns an `NSLClientRef` that your application later uses to prepare NSL lookup requests and to close the NSL session. If no other application has opened a session, calling `NSLOpenNavigationAPI` initializes the NSL Manager. You must call `NSLOpenNavigationAPI` before you call any other NSL Manager functions.

The version of the NSL Manager that comes with the NSL SDK requires Mac OS version 8.0 or later and Open Transport 1.2 or later in order to initialize successfully.

**Note**
The `NSLOpenNavigationAPI` function is a synchronous call. ◆

## NSLCloseNavigationAPI

Close a session with the NSL Manager.

```
void NSLCloseNavigationAPI (NSLClientRef theClient);
```

`theClient` On input, the `NSLClientRef`, obtained by previously calling `NSLOpenNavigationAPI` (page 2-32), that identifies the session that is to be closed.

### DISCUSSION

The `NSLCloseNavigationAPI` function closes the specified NSL Manager session.

▲ **WARNING**
If your application calls `NSLCloseNavigationAPI` while a lookup is in progress, any data that would have been returned is lost. ▲

Your application is responsible for reclaiming memory that it allocates for services lists, parameter blocks, and lookup requests. Your application should reclaim this memory by calling `NSLDisposeServicesList` (page 2-45), `NSLDeleteRequest` (page 2-46) and `NSLFreeTypedDataPtr` (page 2-49), respectively.

**Note**
The `NSLCloseNavigationAPI` is a synchronous call. ◆

# Making a Lookup Request

## NSLMakeNewServicesList

Create a services list.

```
NSLServicesList NSLMakeNewServicesList (char* initialServiceList);
```

`initialServiceList`
: On input, a pointer to a comma-delimited, null-terminated string of service names, such as `http,ftp`.

*function result* A services list. `NSLMakeNewServicesList` returns `NULL` if it can't create the services list because, for example, there is not enough memory or because the NSL Manager is not initialized.

### DISCUSSION

The `NSLMakeNewServicesList` function creates a services list and fills it with the names of the services specified in `initialServiceList`. After you create the services list, you can add the names of additional services by calling `NSLAddServiceToServicesList` (page 2-34).

When you have no further use for the services list, you can reclaim the memory allocated to it by calling `NSLDisposeServicesList` (page 2-45).

**Note**
The `NSLMakeNewServicesList` function is a synchronous call. ◆

## NSLAddServiceToServicesList

Add the name of a service to a services list.

```
NSLError NSLAddServiceToServicesList (NSLServicesList serviceList,
                                      NSLServiceType serviceType);
```

serviceList On input, a services list previously created by calling NSLMakeNewServicesList (page 2-34).

serviceType On input, a service type that is to be added to the services list.

*function result* If the value of NSLError.theErr is noErr, the service was added to the list. Other possible values are kNSLNotInitialized, kNSLBadServiceTypeErr, kNSLNullListPtr, and kNSLBadProtocolTypeErr.

**DISCUSSION**

The NSLAddServicesToServiceList function adds the name of the specified service to a services list.

**IMPORTANT**

You must create serviceList by calling NSLMakeNewServicesList before you call NSLAddServicesToServicesList. ▲

Call NSLAddServiceToServicesList for each service that you want to add to the services list.

**Note**

The NSLAddServiceToServicesList function is a synchronous call. ◆

## NSLPrepareRequest

Create a lookup request.

```
NSLError NSLPrepareRequest (NSLClientNotifyProcPtr notifier,
                     void* contextPtr,
                     NSLClientRef theClient,
                     NSLRequestRef* ref,
                     char* bufPtr,
                     unsigned long bufLen,
                     ClientAsyncInfoPtr* infoPtr);
```

`notifier` On input, `NULL` (for synchronous lookups) or a pointer to your application's notification routine (for asynchronous lookups).

`contextPtr` On input, an application-defined value that your application can use to associate lookup results with the request that initiated them.

`theClient` On input, an `NSLClientRef` obtained by previously calling `NSLOpenNavigationAPI` (page 2-32) that identifies the NSL Manager session.

`ref` On output, a pointer to the resulting lookup request.

`bufPtr` On output, a pointer to the buffer in which lookup results are to be placed.

`bufLen` On output, the length of the buffer pointed to by `bufPtr`.

`infoPtr` On output, `infoPtr` contains default information about how the search is to be conducted. Your application can change the defaults before it starts the lookup.

*function result* If the value of `NSLError.theErr` is `noErr`, the request was created. Other possible values include `kNSLNotInitialized`, `kNSLDuplicateSearchInProgress`, and `kNSLBadClientInfoPtr`.

**DISCUSSION**

The `NSLPrepareRequest` function creates a lookup request, which your application later uses as a parameter when it calls `NSLStartNeighborhoodLookup` (page 2-37) or `NSLStartServicesLookup` (page 2-40).

If `notifier` is null when you call `NSLPrepareRequest`, any lookup that uses the resulting lookup request is performed synchronously. `NSLStartServicesLookup` (page 2-40) and `NSLContinueLookup` (page 2-42) will return when the result buffer is full, the lookup is complete, or an error occurs. Your application can cause `NSLStartServicesLookup` and `NSLContinueLookup` to return at a specified interval, when a specified number of items is in the result buffer, or when a specified amount of time has elapsed by modifying the value of the `alertInterval`, `alertThreshold`, and `maxSearchTime` fields, respectively, of the `ClientAsyncInfo` structure (page 2-27) pointed to by `infoPtr`.

If `notifier` is a pointer to your application's notification routine, your application's notification routine will be called when the result buffer is full, when the lookup is complete, or when an error occurs. Your application can cause your application's notification routine to be called at a specified interval, when a specified number of items is in the result buffer, or when a specified amount of time has elapsed by modifying the value of the `alertInterval`, `alertThreshold`, and `maxSearchTime` fields, respectively, of the `ClientAsyncInfo` structure (page 2-27) pointed to by `infoPtr`.

When your application no longer needs the lookup request, it should call `NSLDeleteRequest` (page 2-46) to reclaim memory associated with the request.

**Note**
The `NSLPrepareRequest` function is a synchronous call. ◆

If `NSLPrepareRequest` returns `kDuplicateSearchInProgress`, there is an ongoing lookup that is using an identical `NSLRequestRef`. Your application can ignore this warning, delete the newly created `NSLRequestRef`, or cancel the lookup that is using the identical `NSLRequestRef`.

## Looking for Neighborhoods and Services

### NSLStartNeighborhoodLookup

Look for neighborhoods.

```
NSLError NSLStartNeighborhoodLookup (NSLRequestRef ref,
                        NSLNeighborhood neighborhood,
                        ClientAsyncInfo* asyncInfo);
```

`ref` On input, an `NSLRequestRef` created by previously calling `NSLPrepareRequest` (page 2-35).

`neighborhood` On input, an `NSLNeighborhood` value created by previously calling `NSLMakeNewNeighborhood` (page 2-51). If `neighborhood` was created with a value of `name` that was `NULL`, `NSLStartNeighborhoodLookup` returns the first default neighborhood. If `neighborhood` was created with a value of `name` that is a name, `NSLStartNeighborhoodLookup` returns a related name. For example, if neighborhood was created with a value of name that is apple.com, NSLStartNeighborhoodLookup returns a subdomain of apple.com.

asyncInfo
On input, a pointer to the asyncInfo structure in whose resultBuffer field NSLStartNeighborhood is to store neighborhood lookup results.

*function result*
If the value of NSLError.theErr is noErr, NSLStartNeighborhoodLookup returned successfully. Possible errors are kNSLNotInitialized, kNSLSearchAlreadyInProgress, kNSLNoPluginsForSearch, kNSLBufferTooSmallForData, and kNSLNullNeighborhoodPtr.

### DISCUSSION

The NSLStartNeighborhoodLookup function returns a neighborhood value that your application can use to define the scope of a subsequent service lookup.

**IMPORTANT**

For any NSLRequestRef, only one neighborhood or service lookup can be in progress at any one time. ▲

If ref was created with a value of notifier that is null, NSLStartNeighborhoodLookup operates synchronously. If ref was created with a value of notifier that is pointer to your application’s notification routine, NSLStartNeighborhoodLookup operates asynchronously.

When NSLStartNeighborhoodLookup returns (if called synchronously) or when the NSL Manager calls your application’s notification routine (if NSLStartNeighborhoodLookup is called asynchronously), your application should check the value of asyncInfo.searchState, which contains one of the following values:

```
kNSLSearchStateBufferFull = 1,
kNSLSearchStateOnGoing = 2,
kNSLSearchStateComplete = 3,
kNSLSearchStateStalled = 4
```

If the value of asyncInfo.searchState is kNSLSearchStatusBufferFull, your application should process the data returned in asyncInfo.resultBuffer. Then it should call NSLContinueLookup (page 2-42) to resume the lookup.

**IMPORTANT**

Calling NSLContinueLookup will cause the information in the result buffer to be overwritten. ▲

If the value of asyncInfo.searchState is kNSLSearchStateOnGoing, the value of asyncInfo.alertInterval or asyncInfo.alertThreshold has been reached. Your application should process the data returned in asyncInfo.resultBuffer. Then it should call NSLContinueLookup to resume the lookup.

If the value of asyncInfo.searchState is kNSLSearchStateComplete, the lookup is complete. Your application should process the data returned in asyncInfo.resultBuffer.

If the value of asyncInfo.searchState is kNSLSearchStateStalled, the value of asyncInfo.alertInterval or asyncInfo.maxSearchTime has been reached, but there is no data in the result buffer. One or more plug-ins for this lookup is waiting to receive data from a server but has not yet timed out. If the value of asyncInfo.searchState is noErr, your application should call NSLContinueLookup to resume the lookup.

If NSLStartNeighborhoodLookup returns kNSLBufferTooSmallForData, the value of asyncInfo.maxBuffserSize is too small to hold an item that would otherwise have been returned. Your application can cancel and restart the lookup, or it can call NSLContinueLookup to resume the lookup even though some data will be lost.

**IMPORTANT**

If more than one plug-in participates in a lookup, the result buffer may contain valid data even though NSLStartNeighborhoodLookup returns an error code from one of the plug-ins. If the value of asyncInfo.searchState is kNSLSearchStateBufferFull, your application should process the data in the result buffer. ▲

**SEE ALSO**

NSLGetNextNeighborhood (page 2-50) for information about processing the data in the result buffer.

## NSLStartServicesLookup

Look for services.

```
NSLError NSLStartServicesLookup (NSLRequestRef ref,
                        NSLNeighborhood neighborhood,
                        TypedDataPtr requestData,
                        ClientAsyncInfo* asyncInfo);
```

`ref` On input, an `NSLRequestRef` created by previously calling `NSLPrepareRequest` (page 2-35).

`neighborhood` On input, an `NSLNeighborhood` value created by previously calling `NSLMakeNewNeighborhood` (page 2-51).

`requestData` On input, a parameter block that describes the search parameters. To format `requestData` properly, call `NSLMakeRequestPB` (page 2-53).

`asyncInfo` On input, a pointer to a `ClientAsyncInfo` structure (page 2-27) obtained by calling `NSLPrepareRequest`.

*function result* If the value of `NSLError.theErr` is `noErr`, `NSLStartServicesLookup` returned successfully. Other possible values are `kNSLNotInitialized`, `kNSLSearchAlreadyInProgress`, `kNSLNoPluginsForSearch`, `kNSLNullNeighborhoodPtr`, and `kNSLBufferTooSmallForData`.

### DISCUSSION

The `NSLStartServicesLookup` function starts a service lookup.

**IMPORTANT**

For any `NSLRequestRef`, only one neighborhood or service lookup can be ongoing at any one time. ▲

If `ref` was created with a value of `notifier` that is null, `NSLStartServicesLookup` operates synchronously. If `ref` was created with a value for `notifier` that is pointer to your application’s notification routine, `NSLStartServicesLookup` operates asynchronously.

When `NSLStartServicesLookup` returns (if called synchronously) or when the NSL Manager calls your application’s notification routine (if

NSLStartServicesLookup is called asynchronously), your application should check the value of asyncInfo.searchState, which contains one of the following values:

```
kNSLSearchStateBufferFull = 1,
kNSLSearchStateOnGoing = 2,
kNSLSearchStateComplete = 3,
kNSLSearchStateStalled = 4
```

If the value of asyncInfo.searchState is kNSLSearchStatusBufferFull, your application should process the data returned in asyncInfo.resultBuffer. Then it should call NSLContinueLookup (page 2-42) to resume the lookup.

**IMPORTANT**

Calling NSLContinueLookup will cause the information in the result buffer to be overwritten. ▲

If the value of asyncInfo.searchState is kNSLSearchStateOnGoing, the value of asyncInfo.alertInterval or asyncInfo.alertThreshold has been reached. Your application should process the data returned in asyncInfo.resultBuffer. Then it should call NSLContinueLookup to resume the lookup.

If the value of asyncInfo.searchState is kNSLSearchStateComplete, the lookup is complete.Your application should process the data returned in asyncInfo.resultBuffer.

If the value of asyncInfo.searchState is kNSLSearchStateStalled, the value of asyncInfo.alertInterval or asyncInfo.maxSearchTime has been reached, but there is no data in the result buffer. One or more plug-ins for this lookup is waiting to receive data from a server but has not yet timed out. If the value of asyncInfo.searchState is noErr, your application should call NSLContinueLookup to resume the lookup.

If NSLStartServicesLookup returns kNSLBufferTooSmallForData, the value of asyncInfo.maxBuffserSize is too small to hold an item that would otherwise have been returned. Your application can cancel and restart the lookup, or it can call NSLContinueLookup to resume the lookup even though some data will be lost.

**IMPORTANT**

If more than one plug-in participates in a lookup, the result buffer may contain valid data even though `NSLStartServicesLookup` returns an error code from one of the plug-ins. If the value of `asyncInfo.searchState` is `kNSLSearchStateBufferFull`, your application should process the data in the result buffer. ▲

To cancel an ongoing lookup, call `NSLCancelRequest` (page 2-45).

**SEE ALSO**

`NSLGetNextUrl` (page 2-51) for information about processing the data in the result buffer. `NSLDeleteRequest` (page 2-46) for information about deleting a lookup request that is no longer needed.

## NSLContinueLookup

Continue a lookup.

```
NSLError NSLContinueLookup (ClientAsyncInfo* asyncInfo);
```

`asyncInfo` A pointer to the `ClientAsyncInfo` structure (page 2-27) for this lookup.

*function result* If the value of `NSLError.theErr` is `noErr`, `NSLContinueLookup` returned successfully. Possible errors include `kNSLNotInitialized`, `kNSLNoContextAvailable`, `kNSLBadClientInfoPtr`, and `kNSLCannotContinueLookup`, and `kNSLBufferTooSmallForData`.

**DISCUSSION**

The `NSLContinueLookup` function continues a service lookup or a neighborhood lookup that has paused because `NSLStartNeighborhoodLookup`, `NSLStartServicesLookup`, or a previous call to `NSLContinueLookup` has returned, or because your application's notification routine has been called. Your application should check the value of `asyncInfo.searchState`, which contains one of the following values:

```
kNSLSearchStateBufferFull = 1,
kNSLSearchStateOnGoing = 2,
kNSLSearchStateComplete = 3,
kNSLSearchStateStalled = 4
```

If the value of `asyncInfo.searchState` is `kNSLSearchStatusBufferFull`, your application should process the data returned in `asyncInfo.resultBuffer`. Then it should call `NSLContinueLookup` again to resume the lookup.

**IMPORTANT**

Calling `NSLContinueLookup` will cause the information in the result buffer to be overwritten. ▲

If the value of `asyncInfo.searchState` is `kNSLSearchStateOnGoing`, the value of `asyncInfo.alertInterval` or `asyncInfo.alertThreshold` has been reached. Your application should process the data returned in `asyncInfo.resultBuffer`. Then it should call `NSLContinueLookup` again to resume the lookup.

If the value of `asyncInfo.searchState` is `kNSLSearchStateComplete`, the lookup is complete.Your application should process the data returned in `asyncInfo.resultBuffer`.

If the value of `asyncInfo.searchState` is `kNSLSearchStateStalled`, the value of `asyncInfo.alertInterval` or `asyncInfo.maxSearchTime` has been reached, but there is no data in the result buffer. One or more plug-ins for this lookup is waiting to receive data from a server but has not yet timed out. If the value of `asyncInfo.searchState` is `noErr`, your application should call `NSLContinueLookup` again to resume the lookup.

If `NSLContinueLookup` returns `kNSLBufferTooSmallForData`, the value of `asyncInfo.maxBuffserSize` is too small to hold an item that would otherwise have been returned. Your application can cancel and restart the lookup, or it can call `NSLContinueLookup` again to resume the lookup even though some data will be lost.

**IMPORTANT**

If more than one plug-in participates in a lookup, the result buffer may contain valid data even though `NSLContinueLookup` returns an error code from one of the plug-ins. If the value of `asyncInfo.searchState` is `kNSLSearchStateBufferFull`, your application should process the data in the result buffer. ▲

To cancel an ongoing lookup, call `NSLCancelRequest` (page 2-45).

**SEE ALSO**

NSLGetNextUrl (page 2-51) for information about processing the data in the result buffer. NSLDeleteRequest (page 2-46) for information about deleting a lookup request that is no longer needed.

## NSLErrorToString

Obtain information about an error.

```
OSStatus NSLErrorToString (NSLError theErr,
                           char* errorString,
                           char* solutionString);
```

theErr — On input, an NSLError structure (page 2-29) whose theErr field contains an NSL error number.

errorString — On input, a pointer to the buffer in which NSLErrorToString is to place a string containing a description of the problem that caused the error. The length of errorString should be 256 bytes.

solutionString — On input, a pointer to the buffer in which NSLErrorToString is to place a string containing a possible solution to the problem. The length of solutionString should be 256 bytes.

*function result* — A value of noErr indicates that NSLErrorToString returned successfully. If NSLError.theContext is zero and NSLError.theErr contains an error number that is not within the range of NSL error numbers, NSLErrorToString returns kNSLBadReferenceErr. The NSLErrorToString function returns kNSLNotInitialized if your application has not opened a session with the NSL Manager by previously calling NSLOpenNavigationAPI (page 2-32).

**DISCUSSION**

The NSLErrorToString function obtains information about an NSLError structure (page 2-29) so that your application can display an appropriate error message. The NSLError structure may have been returned by the NSL Manager or by an NSL plug-in. For any given lookup, search results may be returned by more than one plug-in. You may not want to display an error message if one or more plug-ins return data without error.

**Note**
The `NSLErrorToString` function is a synchronous call. ◆

## NSLCancelRequest

Cancel an ongoing lookup.

```
NSLError NSLCancelRequest (NSLRequestRef ref);
```

`ref` On input, the `NSLRequestRef` obtained by previously calling `NSLPrepareRequest` (page 2-35) for the lookup that is to be canceled.

*function result* If the value of `NSLError.theErr` is `noErr`, the request was canceled successfully. Other possible values are `kNSLNotInitialized` and `kNSLBadReferenceErr`.

**DISCUSSION**

The `NSLCancelRequest` function cancels an ongoing lookup. Any outstanding I/O is also canceled.

# Managing Memory

## NSLDisposeServicesList

Dispose of a services list.

```
void NSLDisposeServicesList (NSLServicesList theList);
```

`theList` On input, the services list that is to be disposed of.

DISCUSSION

The NSLDisposeServicesList function reclaims memory by disposing of a services list. Once you've incorporated the information in a services list into a request parameter block, you can dispose of the services list.

Calling NSLCloseNavigationAPI (page 2-33) does not reclaim memory allocated for services lists, so your application should dispose of services lists before it closes the NSL session.

**Note**
The NSLDisposeServicesList function is a synchronous call. ◆

## NSLDeleteRequest

Delete a lookup request.

```
NSLError NSLDeleteRequest (NSLRequestRef ref);
```

ref — On input, the NSLRequestRef obtained by previously calling NSLPrepareRequest (page 2-35) for the lookup request that is to be deleted.

*function result* — If the value of NSLError.theErr is noErr, the lookup request was deleted. Other possible values are kNSLNotInitialized and kNSLBadReferenceErr.

DISCUSSION

The NSLDeleteRequest function deletes the specified lookup request and deallocates memory associated with it, including the ClientAsyncInfo structure. If a lookup is in progress for the specified lookup request when you call NSLDeleteRequest, the lookup is terminated and any outstanding I/O is lost.

The NSLDeleteRequest function does not deallocate memory associated with the services list or request parameter blocks. To deallocate memory for services lists, call NSLDisposeServicesList (page 2-45); to deallocate memory for parameter blocks, call NSLFreeTypedDataPtr (page 2-49).

**Note**
The NSLDeleteRequest is a synchronous call. ◆

# Managing Services

## NSLRegisterService

Register a service.

```
NSLError NSLRegisterService (TypedDataPtr requestData);
```

requestData — On input, a TypedDataPtr containing information about the service that is to be registered. To format requestData properly, call NSLMakeRegistrationPB (page 2-52).

*function result* — If the value of NSLError.theErr is noErr, the service was registered. The NSLRegisterService function returns kNSLNoSupportForService, which indicates that none of the currently installed plug-ins support the service for which registration is requested. Another possible error is kNSLNotInitialized.

### DISCUSSION

The NSLRegisterService function registers the specified service with the NSL Manager. An application that provides a network service should call NSLRegisterService as part of its standard startup procedure.

Once the service is registered, your application can call NSLFreeTypedDataPtr (page 2-49) to reclaim memory allocated for requestData.

**Note**
The NSLRegisterService is a synchronous call. ◆

Your application should keep its NSL Manager session open while the services it registers remain available. Your application should call NSLDeregisterService (page 2-48) to deregister its services as part of its standard shutdown procedure.

Then your application can call `NSLCloseNavigationAPI` (page 2-33) to close its NSL session.

## NSLDeregisterService

Deregister a service.

```
NSLError NSLDeregisterService (TypedDataPtr requestData);
```

`requestData` On input, a `TypedDataPtr` identifying the service that is to be deregistered. You can use the `requestData` value that you used to register the service (if you have not already disposed of it) or you can call `NSLMakeRegistrationPB` (page 2-52) to format `requestData` properly.

*function result* If the value of `NSLError.theErr` is `noErr`, the service was deregistered. The `NSLDeregisterService` function returns `kNSLNoSupportForService`, which indicates that none of the currently installed plug-ins support the service for which deregistration is requested. Other possible errors include `kNSLNotInitialized`.

### DISCUSSION

The `NSLDeregisterService` function deregisters the service specified by `requestData`. You should call `NSLDeregisterService` as part of your standard shutdown procedure. Once your application has deregistered the services it has previously registered, it can call `NSLCloseNavigationAPI` (page 2-33).

**Note**

The `NSLDeregisterService` is a synchronous call. ◆

# NSL Manager Utility Functions

## NSLFreeNeighborhood

Dispose of an `NSLNeighborhood` value.

```
NSLNeighborhood NSLFreeNeighborhood (NSLNeighborhood neighborhood);
```

`neighborhood` On input, the `NSLNeighborhood` value that is to be disposed of.

*function result* If no error occurred, an `NSLNeighborhood` value whose value is `NULL`. Possible errors include `kNSLBadDataTypeErr`, which indicates that `neighborhood` is not of the type `NSLNeighborhood`.

**DISCUSSION**

The `NSLMakeNewNeighborhood` function disposes of an `NSLNeighborhood` value and reclaims that memory that was allocated to it.

**Note**
The `NSLMakeNewNeighborhood` function is a synchronous call. ◆

## NSLFreeTypedDataPtr

Free memory allocated for a request or registration parameter block.

```
TypedDataPtr NSLFreeTypedDataPtr (TypedDataPtr nslTypeData);
```

`nslTypeData` On input, a `TypedDataPtr` obtained by previously calling `NSLMakeRequestPB` (page 2-53) or `NSLMakeRegistrationPB` (page 2-52).

*function result* If no error occurred, a `TypedDataPtr` whose value is `NULL`. Possible errors include `kNSLBadDataTypeErr`, which indicates that `nslTypeData` is not a valid parameter block.

**DISCUSSION**

The `NSLFreeTypedDataPtr` function frees memory that your application caused to be allocated when it previously called `NSLMakeRequestPB` (page 2-53) and `NSLMakeRegistrationPB` (page 2-52). Your application should call `NSLFreeTypedDataPtr` (page 2-49) when it has no further use for the parameter block specified by `nslTypeData`.

## NSLGetNextNeighborhood

Obtain information about the next neighborhood in a buffer.

```
Boolean NSLGetNextNeighborhood (ClientAsyncInfoPtr infoPtr,
                        NSLNeighborhood* neighborhood,
                        long* neighborhoodlength);
```

`infoPtr` On input, a pointer to a `ClientAsyncInfo` structure (page 2-27) whose `resultBuffer` field may contain another neighborhood.

`neighborhood` On output, if another neighborhood was found in `resultBuffer`, a pointer to the beginning of the neighborhood's name.

`neighborhoodLength`
On output, the length of the neighborhood pointed to by `neighborhood`.

*function result* A Boolean value. A value of `TRUE` indicates that `neighborhood` points to the next neighborhood in `resultBuffer`. A value of `FALSE` indicates that there are no more names in `resultBuffer`.

**DISCUSSION**

The `NSLGetNextNeighborhood` function obtains the starting position and the length of the next neighborhood in a result buffer.

**Note**
The NSLGetNextNeighborhood function is a synchronous call. ◆

## NSLGetNextUrl

Obtain information about the next URL in a buffer.

```
Boolean NSLGetNextUrl (ClientAsyncInfoPtr infoPtr,
                          char** urlPtr,
                          long* urlLength);
```

infoPtr — On input, a pointer to a ClientAsyncInfo structure (page 2-27) whose resultBuffer field may contain a URL.

urlPtr — On output, if a URL was found in resultBuffer, a pointer to the beginning of the URL.

urlLength — On output, the length of the URL pointed to by urlPtr.

*function result* — A Boolean value. A value of TRUE indicates that urlPtr points to the next URL in resultBuffer. A value of FALSE indicates that there are no more URLs in resultBuffer.

### DISCUSSION

The NSLGetNextUrl function obtains the starting position and the length of the next URL in a result buffer. You call NSLGetNextUrl to parse the URLs returned by a previous call to NSLStartServicesLookup.

## NSLMakeNewNeighborhood

Create a neighborhood.

```
NSLNeighborhood NSLMakeNewNeighborhood (char* name,
                          char* protocolList);
```

`name` On input, a pointer to a string containing a name. If `dns` is specified in `protocolList`, the value of `name` should be a domain name, such as `apple.com`. If `slp` is specified in `protocolList`, the value of name should be a scope. Other types of names may be appropriate depending on the installed plug-ins. To create an `NSLNeighborhood` that can be used to obtain a list of default neighborhoods when you call `NSLStartNeighborhoodLookup`, set `name` to `NULL`.

`protocolList` On input, a pointer to a comma-separated, null-terminated list of protocols (such as `dns,slp`) that are to participate in a lookup conducted with the resulting `NSLNeighborhood` value. If the value of `protocolList` is `NULL`, all available protocols will participate in the lookup.

*function result* An `NSLNeighborhood` value that can be used in a subsequent call to `NSLStartServicesLookup`. If `NSLMakeNewNeighborhood` can't create an `NSLNeighborhood` value, it returns `NULL`. This might happen, for example, if there is not enough memory.

**DISCUSSION**

The `NSLMakeNewNeighborhood` function creates an `NSLNeighborhood` value that defines the boundary of a subsequent search.

When you have no further use for an `NSLNeighborhood` value, you can reclaim the memory allocated to it by calling `NSLFreeNeighborhood` (page 2-49).

**Note**
The `NSLMakeNewNeighborhood` function is a synchronous call. ◆

## NSLMakeRegistrationPB

Create a registration parameter block.

```
OSStatus NSLMakeRegistrationPB (NSLAttribute attribute,
                    NSLPath urlToService,
                    TypedDataPtr* newDataPtr);
```

`attribute` On input, a null-terminated string containing a description of the service that is being registered, such as “Web Sharing.” The value of `attribute` differentiates the service from other services of the same type.

`urlToService` On input, a null-terminated string containing the URL of the service that is to be registered. The URL, such as `http://www.apple.com`, includes the service type (in this example, `http`).

`newDataPtr` On input, the address of the `TypedDataPtr` at which `NSLMakeRegistrationPB` is to place the resulting parameter block.

*function result* A value of `noErr` indicates that `NSLMakeRegistrationPB` returned successfully. A possible error is `kNSLNotInitialized`**.**

**DISCUSSION**

The `NSLMakeRegistrationPB` function creates a parameter block that is formatted properly for use with subsequent calls to `NSLRegisterService` and `NSLDeregisterService` (page 2-48).

## NSLMakeRequestPB

Create a request parameter block.

```
OSStatus NSLMakeRequestPB (NSLServicesList serviceList,
                           NSLAttribute attribute,
                           TypedDataPtr* newDataPtr);
```

`serviceList` On input, an `NSLServiceList` created by previously calling `NSLMakeNewServicesList` (page 2-34). If `serviceList` is empty, all possible services will be returned in the lookup.

`attribute` On input, a null-terminated string containing a description of the service that is to be searched for, such as “Web Sharing.” The value of `attribute` differentiates one service from other services of the same type. If `attribute` is empty, any subsequent lookup will not differentiate between services of the same type.

newDataPtr On input, the address of the TypedDataPtr at which NSLMakeRequestPB is to place the resulting parameter block.

*function result* A value of noErr indicates that NSLMakeRequestPB returned successfully. Possible errors include kNSLBadDomainErr.

**DISCUSSION**

The NSLMakeRequestPB function creates a parameter block that is formatted properly for use with subsequent calls to NSLStartServicesLookup (page 2-40).

# NSL Manager Result Codes

All of the NSL Manager functions return a result code. The result codes specific to the NSL Manager are listed here. In addition, NSL Manager functions may return other Mac OS result codes, which are described in *Inside Macintosh.*

| | | |
|---|---|---|
| noErr | 0 | No error. |
| kNSLNotInitialized | -4199 | The NSL Manager could not be initialized. |
| kNSLInsufficientSysVer | -4198 | The installed version of the Mac OS does not support the NSL Manager. (For the NSL Manager SDK, Version 8.0 or later is required.) |
| kNSLInsufficientOTVer | -4197 | The installed version of Open Transport does support the NSL Manager. (Open Transport 1.2 or later is required.) |
| kNSLNoElementsInList | -4196 | A specified list is empty. |
| kNSLBadReferenceErr | -4195 | The specified NSLClientRef or NSLRequestRef is invalid. |
| kNSLBadServiceTypeErr | -4194 | The specified service type is not supported. |

| | | |
|---|---|---|
| `kNSLBadDataTypeErr` | -4193 | The specified parameter is not of the correct data type. |
| `kNSLBadNetConnection` | -4192 | A network error occurred. AppleTalk or TCP/IP may be turned off, or the computer may not be connected to the network. |
| `kNSLNoSupportForService` | -4191 | No plug-in supports the requested service registration or deregistration. |
| `kNSLInvalidPluginSpec` | -4190 | The `theContext` field of the specified `NSLError` structure is invalid. |
| `kNSLMismatchedBufferLengths` | -4189 | Reserved. |
| `kNSLRequestBufferAlreadyInList` | -4188 | Reserved. |
| `kNSLNoContextAvailable` | -4187 | The `asyncInfo` parameter provided in a call to `NSLContinueLookup` is invalid. |
| `kNSLBufferTooSmallForData` | -4186 | The application's result buffer is too small to store the data returned by a plug-in. |
| `kNSLCannotContinueLookup` | -4185 | The lookup cannot be continued due to an error condition or a bad state. |
| `kNSLBadClientInfoPtr` | -4184 | The specified `ClientAsyncInfoPtr` is invalid. |
| `kNSLNullListPtr` | -4183 | The pointer to the specified list is invalid. |
| `kNSLBadProtocolTypeErr` | -4182 | The specified `NSLServiceType` is empty. |
| `kNSLPluginLoadFailed` | -4181 | During system initialization, the NSL Manager was unable to load any plug-ins. |
| `kNSLNoPluginsFound` | -4180 | During system initialization, the NSL Manager was unable to find any valid plug-ins to load. |

| | | |
|---|---|---|
| `kNSLSearchAlreadyInProgress` | -4179 | A search is already in progress for the specified `clientRef`. |
| `kNSLNoPluginsForSearch` | -4178 | None of the installed plug-ins are able to respond to the lookup request. |
| `kNSLNullNeighborhoodPtr` | -4177 | The pointer to a neighborhood is invalid. |
| `kNSLSomePluginsFailedToLoad` | -4176 | During system initialization, the NSL Manager was unable to load some plug-ins. |
| `kNSLErrNullPtrError` | -4175 | A specified pointer is invalid. |
| `kNSLNotImplementedYet` | -4174 | The requested functionality is not available. |

CHAPTER 3

# Network Services Location Manager Plug-In Reference

## Contents

# NSL Manager Plug-in Constants and Data Types

## PluginAsyncInfo Structure

The NSL Manager passes a `PluginAsyncInfo` structure as a parameter to the plug-in’s `StartNeighborhoodLookup`, `StartServicesLookup`, and `ContinueLookup` routines. The `PluginAsyncInfo` structure contains all of the information that the plug-in needs to start or continue a lookup. The plug-in uses the `PluginAsyncInfo` structure to maintain state information about an ongoing lookup request and to return information about the lookup to the NSL Manager.

```
typedef struct PluginAsyncInfo
{
    void*           mgrContextPtr;
    void*           pluginContextPtr;
    void*           pluginPtr;
    char*           resultBuffer;
    long            bufferLen;
    long            maxBufferSize;
    UInt32          maxSearchTime;
    UInt32          reserved1;
    UInt32          reserved2;
    UInt32          reserved3;
    NSLCLientRef    clientRef
    NSLRequestRef   requestRef
    NSLSearchState  searchState;
    OSStatus        searchResult;
} PluginAsyncInfo, *PluginAsyncInfoPtr;
```

**Field descriptions**

`mgrContextPtr` A value set by the NSL Manager for its own use.

`pluginContextPtr` A value set by the plug-in for its own use.

`pluginPtr` A pointer to the plug-in object that is waiting for the application to call `NSLContinueLookup`.

resultBuffer
A pointer to the buffer that the plug-in can use to store lookup results.

bufferLen
The length of valid data in resultBuffer.

maxBufferSize
The maximum length of resultBuffer.

maxSearchTime
The total amount of time, specified in ticks, that is to be spent on the lookup. The default value is zero, which indicates that the lookup time is not to be limited. An application may specify a maximum search time before it calls NSLStartNeighborhoodLookup or NSLStartServicesLookup, in which case the plug-in is obligated to comply. The NSL Manager does not have a mechanism for enforcing compliance.

Reserved1
Reserved.

Reserved2
Reserved.

Reserved3
Reserved.

clientRef
A value identifying the application that made the request.

requestRef
A value specifying the lookup request.

searchState
A value that the plug-in sets to indicate the current state of the lookup. The value can be one of the following:

```
kNSLSearchStateBufferFull= 1,
kNSLSearchStateOnGoing = 2,
kNSLSearchStateComplete = 3,
kNSLSearchStateStalled = 4
```

searchResult
An NSLError structure that the plug-in uses to return error information.

## PluginData Structure

Plug-ins use the PluginData structure to inform the NSL Manager about the protocols and services they support. The NSL Manager obtains this information by calling the plug-in's GetPluginInfo routine (page 3-65).

```
typedef struct PluginData
{
    long        reserved1;
    long        reserved2;
    long        reserved3;
    Boolean     isPurgeable;
    UInt16      totalLen;
    UInt16      dataTypeOffset;
    UInt16      serviceListOffset;
    UInt16      protocolListOffset;
    UInt16      commentStringOffset;
} PluginData, *PluginDataPtr;
```

**Field descriptions**

`reserved1` Reserved.

`reserved2` Reserved.

`reserved3` Reserved.

`isPurgeable` `TRUE` if the plug-in can be purged from memory; `FALSE` if the plug-in cannot be purged from memory.

`totalLen` The length of the `PluginData` structure.

`dataTypeOffset` The offset from the beginning of the `PluginData` structure at which the data type resides.

`serviceListOffset` The offset from the beginning of the `PluginData` structure at which are listed the services the plug-in supports.

`protocolListOffset` The offset from the beginning of the `PluginData` structure at which are listed the protocols the plug-in supports.

`commentStringOffset`

The offset from the beginning of the `PluginData` structure at which the comment string begins. The comment string should describe the protocols and services that the plug-in supports, and may contain other information suitable for display to the user.

# NSL Manager Plug-in Utility Functions

## NSLGetNameFromNeighborhood

Obtain the name from a neighborhood.

```
void NSLGetNameFromNeighborhood (NSLNeighborhood neighborhood,
                     char** name,
                     long* length);
```

| | |
|---|---|
| `neighborhood` | On input, the `NSLNeighborhood` value from which the name is to be obtained. |
| `name` | On output, the name that `neighborhood` contains. |
| `length` | On output, the length of `name`. |

**DISCUSSION**

The `NSLGetNameFromNeighborhood` function obtains the name from an `NSLNeighborhood` value. The plug-in uses the name to limit the scope of a lookup. The format of the name is depends on the protocols that the plug-in supports. For example, a plug-in that supports DNS would expect the value of `name` to be a name such as `apple.com`, and an plug-in that supports AppleTalk would expect `name` to be an AppleTalk zone name, such as `CC 6 4th Floor South`.

NSL plug-ins call `NSLGetNameFromNeighborhood` from their `StartNeighborhoodLookup` (page 3-67) and `StartServicesLookup` (page 3-68) routines.

## NSLParseRegistrationPB

Parse a registration parameter block.

```
OSStatus NSLParseRegistrationPB (TypedDataPtr newDataPtr,
                    char** attributePtr,
                    UInt16* attributeLen,
                    char** urlPtr,
                    UInt16* urlLen);
```

`newDataPtr` On input, the registration parameter block that is to be parsed.

`attributePtr` On output, a pointer to the attribute stored in a registration parameter block.

`attributeLen` On output, a pointer to the length of the attribute pointed to by `attributePtr`.

`urlPtr` On output, a pointer to the URL stored in a registration parameter block.

`urlLen` On output, a pointer to the length of the URL pointed to by `urlPtr`.

*function result* A value of `noErr` indicates that `NSLParseRequestPB` returned successfully. The `NSLParseRegistrationPB` function returns `kBadDataTypeError` if `newDataPtr` is not a valid request parameter block.

**DISCUSSION**

The `NSLParseRegistrationPB` function parses a registration parameter block. When the NSL Manager calls an NSL plug-in’s `Register` routine (page 3-66), the NSL Manager passes the parameter block to the plug-in, which calls `NSLParseRegistrationPB` to determine the location of the list of URLs and attributes the application has specified for the registration.

### NSLParseRequestPB

Parse a request parameter block.

```
OSStatus NSLParseRequestPB (TypedDataPtr newDataPtr,
                        UInt16* serviceListOffset,
                        UInt16* attributeOffset);
```

`newDataPtr` On input, the address of the `TypedDataPtr` that is to be parsed.

`serviceListOffset`
On output, the offset in `newDataPtr` at which the service list begins.

`attributeOffset`
On output, the offset in `newDataPtr` at which the attribute begins.

*function result* A value of `noErr` indicates that `NSLParseRequestPB` returned successfully. The `NSLParseRequestPB` function returns `kBadDataTypeError` if `newDataPtr` is not a valid request parameter block.

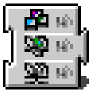
**DISCUSSION**

The `NSLParseRequestPB` function parses a request parameter block. When the NSL Manager calls an NSL plug-in’s `StartServicesLookup` routine (page 3-68), the NSL Manager passes the parameter block to the plug-in, which calls `NSLParseRequestPB` to determine the location of the service list and attributes that the application has specified for the lookup.

## NSL Manager Plug-in Routines

NSL plug-ins reside in the Extensions folder inside the System Folder. The icon for NSL plug-ins is shown in Figure 3-1.

**Figure 3-1** NSL plug-in icon



The creator code for an NSL plug-in is `'NSLp'` and the type code is `'shlb'`.

Each NSL plug-in provides the routines described in this section for the NSL Manager to call.

## GetPluginInfo

Provide information about the plug-in.

```
OSStatus GetPluginInfo (PluginDataPtr* theData);
```

`theData` A pointer to a `PluginData` structure (page 3-60) in which the plug-in is to return information about itself.

*result* A value indicating that `GetPluginInfo` completed successfully. The `GetPluginInfo` routine can return any NSL error code to indicate that it did not provide the requested information.

**DISCUSSION**

The NSL Manager calls a plug-in’s `GetPluginInfo` routine to determine the protocols, data types, and services that the plug-in supports. The NSL Manager uses the information returned in `theData` to determine which plug-ins to use when applications attempt to register services or start lookups.

## InitPlugin

Initialize the plug-in.

```
OSStatus InitPlugin (void);
```

*result* A value of `noErr` indicates that `InitPlugin` successfully initialized the plug-in.

**DISCUSSION**

The `InitPlugin` routine allocates memory for the plug-in and opens network connections that the plug-in will use.

## Register

Register services.

```
OSStatus Register (TypedDataPtr dataPtr);
```

`dataPtr` A `TypedDataPtr` that specifies the services that are to be registered.

*result* A value of `noErr` indicates that `Register` successfully registered the specified services. To indicate that the services were not successfully registered, `Register` can return any NSL error code. For example, if `Register` cannot parse `dataPtr`, it should return `kNSLBadDataTypeErr`.

**DISCUSSION**

The `Register` routine registers the services specified by `dataPtr`. The NSL Manager calls a plug-in's `Register` routine in response to an application that calls the NSL Manager's `NSLRegisterService` function (page 2-47).

To parse the services specified by `dataPtr`, the `Register` routine calls `NSLParseRegistrationPB` (page 3-63).

## StartNeighborhoodLookup

Look for neighborhoods.

```
OSStatus StartNeighborhoodLookup (NSLNeighborhood neighborhood,
                        NSLMgrNotifyProcPtr notifier,
                        PluginAsyncInfo* infoPtr);
```

`neighborhood` An `NSLNeighborhood` value that identifies the neighborhood in which the lookup is to be conducted.

`notifier` The NSL Manager's notification routine, which the plug-in calls when the result buffer contains one item, the lookup is complete, the maximum search time has been reached, or an error has occurred.

`infoPtr` A pointer to a `PluginAsyncInfo` structure whose `clientRef` and `requestRef` fields identify the application and request associated with the lookup that is to be started, whose `maxSearchTime` field may limit the amount of time that is to be spent on the lookup, whose `resultBuffer` field is to be changed to point to the plug-in's lookup result, and whose `searchState` and `searchResult` fields are used to store status information about the lookup.

*result* A value of `noErr` indicates that `StartNeighborhoodLookup` completed successfully. The `StartNeighborhoodLookup` routine can return any NSL error code to indicate that it did not start the lookup. A value of `kNSLBadDataTypeErr` indicates that the search was not started because one or more of the input parameters is invalid.

### DISCUSSION

The `StartNeighborhoodLookup` routine performs the lookup specified by `neighborhood`. The NSL Manager calls a plug-in's `StartNeighborhoodLookup` routine in response to an application that calls the NSL Manager's `NSLStartNeighborhoodLookup` function (page 2-37).

To obtain the name of the neighborhood specified by `neighborhood`, the `StartNeighborhoodLookup` routine calls `NSLGetNameFromNeighborhood` (page 3-62).

A plug-in's StartNeighborhoodLookup routine stores one neighborhood in its result buffer, changes infoPtr.resultBuffer to point to its result buffer, sets infoPtr.bufferLen to the length of the valid data in its result buffer, and calls the NSL Manager's notification routine.

If the value of infoPtr.maxSearchTime  is a non-zero positive value when the plug-in's StartNeighborhoodLookup routine is called, StartNeighborhoodLookup routine should maintain a count of the time in ticks that it spends on this lookup.

To maintain context information about this lookup, the StartNeighborhoodLookup routine can use infoPtr.pluginContext.

## StartServicesLookup

Look for services.

```
OSStatus StartServicesLookup (NSLNeighborhood neighborhood,
                        TypedDataPtr dataPtr,
                        NSLMgrNotifyProcPtr notifier,
                        PluginAsyncInfo* infoPtr);
```

neighborhood An NSLNeighborhood value that specifies the neighborhood in which the service lookup is to be conducted.

dataPtr A TypedDataPtr that identifies the parameters for a lookup.

notifier The NSL Manager's notification routine, which the plug-in calls when the result buffer contains one item, the lookup is complete, the maximum search time has been reached, or an error has occurred.

infoPtr A pointer to a PluginAsyncInfo structure whose clientRef and requestRef fields identify the application and request associated with the lookup that is to be started, whose maxSearchTime field may specify a limit on the amount of time that is to be spent on the search, whose resultBuffer field should be changed to point to the plug-in result, and whose searchState and searchResult fields are used to store status information about the lookup.

*result* A value of noErr indicates that StartServicesLookup completed successfully. The StartServicesLookup routine can return any NSL error code to indicate that it did not start the lookup. A value of kNSLBadDataTypeErr indicates that the search was not started because one or more of the input parameters is invalid.

**DISCUSSION**

The StartServicesLookup routine performs the lookup specified by dataPtr. The NSL Manager calls a plug-in's StartServicesLookup routine in response to an application that calls the NSL Manager's NSLStartServicesLookup function (page 2-40).

**IMPORTANT**

The StartServicesLookup routine may be called synchronously or asynchronously, so it should be prepared to handle both modes. ▲

To obtain the name of the neighborhood specified by neighborhood, the StartNeighborhoodLookup routine calls NSLGetNameFromNeighborhood (page 3-62).

To parse the lookup parameters specified by dataPtr, the StartServicesLookup routine calls NSLParseRequestPB (page 3-64).

Upon receipt of a URL, a plug-in's StartServicesLookup routine places the URL in its result buffer, changes infoPtr.resultBuffer to point to its result buffer, sets infoPtr.bufferLen to the length of the valid data in its result buffer, and calls the NSL Manager's notification routine.

If the value of infoPtr.maxSearchTime is a non-zero positive value, StartServicesLookup should limit the overall time for this lookup to the specified time in ticks if the specified time is less than the plug-in's own limit.

To maintain context information about this lookup, the StartServicesLookup routine can use infoPtr.pluginContextPtr.

## ContinueLookup

Continue a lookup.

```
OSStatus ContinueLookup (NSLMgrNotifyProcPtr notifier,
                         PluginAsyncInfo* infoPtr);
```

`notifier` The NSL Manager's notification routine, which the plug-in's `ContinueLookup` routine calls when the result buffer is full, the lookup is complete, the maximum search time, alert interval, or item threshold has been reached, or an error has occurred.

`infoPtr` A pointer to a `PluginAsyncInfo` structure whose `clientRef` and `requestRef` fields identify the application and request associated with the lookup that is to be continued, whose `resultBuffer` field is used to store lookup results, and whose `searchState` and `searchResult` fields are used to store status information about the lookup.

*result* A value indicating that `ContinueLookup` completed successfully. The `ContinueLookup` routine can return any NSL error code to indicate that it did not continue the lookup.

### DISCUSSION

The `ContinueLookup` routine continues a lookup that is in progress. The lookup that is to be continued is identified by `infoPtr.requestRef`. The NSL Manager calls a plug-in's `ContinueLookup` routine in response to an application that calls the NSL Manager's `NSLContinueLookup` function (page 2-42).

**IMPORTANT**

The `ContinueLookup` routine may be called synchronously or asynchronously, so the `ContinueLookup` routine should be prepared to handle both modes. ▲

Upon receipt of an item, a plug-in's `ContinueLookup` routine places the item in its result buffer, changes `infoPtr.resultBuffer` to point to its result buffer, sets `infoPtr.bufferLen` to the length of the valid data in its result buffer, and calls the NSL Manager's notification routine.

If the value of `infoPtr.maxSearchTime` was a non-zero positive value when the plug-in's `StartServicesLookup` routine was called, `ContinueLookup` routine should maintain a count of the time that it has spent on this lookup and limit the search to the specified time.

**Note**
For a particular lookup, the value of `infoPtr.maxSearchTime` should not change between calls to `StartServicesLookup` and `ContinueLookup` or between successive calls to `ContinueLookup`. ◆

## ErrNumToString

Provide strings that correspond to error codes.

```
OSStatus ErrNumToString (OSStatus errNum,
                         StringPtr errString,
                         StringPtr theSolution);
```

`errNum` The error code, previously returned by one of the plug-in's routines, for which a string description and string solution are to be obtained.

`errString` A `StringPtr` in which `ErrorToString` is to place a string of up to 256 bytes that describe the error condition that corresponds to `theError`.

`theSolution` A S`tringPtr` in which `ErrNumToString` is to place a string of up to 256 bytes that suggest a solution to the error condition that corresponds to `errNum`.

*result* A value indicating that `ErrNumToString` completed successfully. The `ErrNumToString` routine can return any NSL error code to indicate that it did not provide the requested strings.

### DISCUSSION

The `ErrNumToString` routine stores in `errString` a string that describes the error number specified by `errNum` and stores in `theSolution` a suggested solution. The NSL Manager calls a plug-in's `ErrNumToString` routine when an application calls the NSL Manager's `NSLErrorToString` function (page 2-44).

When `ErrNumToString` returns, the NSL Manager returns the strings to the application. The application may choose to display `errString` and `theSolution` to the user, so both strings should be suitable for display.

**IMPORTANT**

A plug-in's `ErrNumToString` routine should be able to provide a value for `errString` and `theSolution` for every error code that the plug-in returns. ▲

## CancelLookup

Cancel a lookup.

```
OSStatus CancelLookup (PluginAsyncInfo* infoPtr);
```

`infoPtr` A pointer to a `PluginAsyncInfo` structure whose `requestRef` field identifies the lookup that is to be canceled.

*result* A value indicating that `CancelLookup` successfully canceled the lookup. The `CancelLookup` routine can return any NSL error code to indicate that it did not cancel the lookup.

**DISCUSSION**

The `CancelLookup` routine cancels the lookup associated with `infoPtr.requestRef`. The NSL Manager calls a plug-in's `CancelLookup` routine when an application calls the NSL Manager's `NSLCancelRequest` function (page 2-45).

## Deregister

Deregister services.

```
OSStatus Deregister (TypedDataPtr dataPtr);
```

`dataPtr` A `TypedDataPtr` that identifies the services that are to be deregistered.

*result* A value of noErr indicates that Deregister successfully deregistered the specified services. To indicate that the services were not successfully deregistered, Deregister can return any NSL error code. For example, if Deregister cannot parse dataPtr, it should return kNSLBadDataTypeErr.

**DISCUSSION**

The Deregister routine deregisters the services specified in dataPtr. The NSL Manager calls a plug-in's Deregister routine in response to an application that calls the NSL Manager's NSLDeregisterService function (page 2-48).

To parse the services specified by dataPtr, the Deregister routine calls NSLParseRegistrationPB (page 3-63).

## UnloadPlugin

Unload a plug-in.

```
OSStatus UnloadPlugin(Boolean forceQuit);
```

forceQuit A Boolean value. If forceQuit is TRUE, the UnloadPlugin routine must deinitialize itself completely. If forceQuit is FALSE, the UnloadPlugin routine can conduct all or part of its deinitialization procedures at its discretion.

*result* A value of noErr indicates that UnloadPlugin successfully deinitialized the plug-in. If the value of forceQuit is FALSE, UnloadPlugin can return any NSL error code to indicate that it needs to remain in memory.

**DISCUSSION**

The UnloadPlugin routine stops any of its lookups that are in progress, closes open network connections, and deallocates memory that it has allocated for its use.

The NSL Manager calls a plug-in's UnloadPlugin routine in response to an application that calls the NSL Manager's NSLCloseNavigationAPI function (page 2-33) when that application is the last application having an open NSL Manager session.

If the plug-in needs to remain in memory (for example, to handle requests for registered services) and if `forceQuit` is `FALSE`, the plug-in can return an error code and remain in memory. However, if `forceQuit` is `TRUE`, the plug-in must deinitialize itself completely and prepare to be unloaded from memory.

# Index