Inside Mac OS X

Directory Services

Technical Publications

April 10, 2000. Draft. Preliminary.

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings 9

Preface About This Manual 11

Conventions used in this manual 11
For more information 11

Chapter 1 About Directory Services 13

Attributes 16
Authentication and Authorization 16
Directory Native Authentication 18
Directory Services Metanode 18

Chapter 2 Using Directory Services 21

Manipulating Directory Nodes 21
  Opening a Directory Node 21
  Finding a Directory Node 23
  Listing Directory Nodes 25
  Authenticating Users to Directory Nodes 28
    Clear Text Authentication 28
    APOP Authentication 30
    Native Authentication 33
Manipulating Records 36
  Creating a Record 37
  Getting a List of Records 38
  Obtaining Information About a Record 40
  Setting the Name of a Record 42
  Setting the Type of a Record 43
  Deleting a Record 45
Managing Threads 46

Chapter 3 Directory Services Reference 47

Directory Services Functions 47
Managing Directory Services Sessions 47
dsOpenDirService 48
dsCloseDirService 48
Managing Directory Nodes 49
dsCloseDirNode 49
dsDoSetPassword 50
dsDoDirNodeAuth 52
dsFindDirNodes 54
dsGetDirNodeCount 56
dsGetDirNodeInfo 56
dsGetDirNodeList 58
dsGetDirNodeName 59
dsOpenDirNode 60
Managing Records 61
dsCloseRecord 61
dsCreateRecord 62
dsCreateRecordAndOpen 63
dsDeleteRecord 64
dsFlushRecord 64
dsGetRecordEntry 65
dsGetRecordList 66
dsGetRecordReferenceInfo 68
dsOpenRecord 69
dsSetRecordAccess 70
dsSetRecordFlags 71
dsSetRecordName 71
dsSetRecordType 72
Working with Attributes 73
dsAddAttribute 73
dsAddAttributeValue 74
dsDoAttributeValueSearch 75
dsGetAttributeValue 77
dsGetAttributeEntry 78
dsGetRecordAttributeInfo 80
dsGetRecordAttributeValueByID 81

dsGetRecordAttributeValueByIndex 82
dsRemoveAttribute 83
dsRemoveAttributeValue 83
dsSetAttributeAccess 84
dsSetAttributeFlags 85
dsSetAttributeValue 86
Directory Services Utility Functions 86
Attribute Utility Functions 87
dsAllocAttributeValueEntry 87
dsDeallocAttributeValueEntry 88
Data Buffer Utility Functions 89
dsDataBufferAllocate 89
dsDataBufferDeAllocate 90
Data List Utility Functions 90
dsAppendStringToList 91
dsBuildListFromNodes 92
dsBuildListFromNodesAlloc 93
dsBuildFromPath 94
dsBuildListFromStrings 95
dsBuildListFromStringsAlloc 95
dsDataListAllocate 96
dsDataListCopyList 97
dsDataListDeAllocate 98
dsDataListGetNodeAlloc 99
dsDataListGetNodeCount 99
dsDataListInsertNode 100
dsDataListMergeList 101
dsDataListRemoveNodes 101
dsDataListThisNode 102
dsGetDataLength 103
dsGetPathFromList 103
Data Node Utility Functions 104
dsDataNodeAllocateBlock 104
dsDataNodeAllocateString 105
dsDataNodeDeAllocate 106
dsDataNodeGetSize 107
dsDataNodeSetLength 107
Miscellaneous Utility Functions 108

dsDoPluginCustomCall 108
dsReleaseContinueData 109
Managing Custom Routines 110
dsGetCustomAllocate 110
dsRegisterCustomMemory 112
dsUnRegisterCustomMemory 113
dsGetCustomThread 114
dsRegisterCustomThread 115
dsUnRegisterCustomThread 116
Application-Defined Routines 117
Custom Memory Routines 117
Custom Thread Routines 118
Directory Services Structures and Other Data Types 118
tDirReference 119
tDirNodeReference 119
tClientData 120
tBuffer 120
tContextData 120
tDataBuffer 121
tDataNode 121
tDataList 122
tAccessControlEntry 122
tAttributeValueEntry 123
tAttributeEntry 123
tRecordEntry 124
Directory Services Constants 125
Well Known Attribute Type Constants 125
Pattern-Matching Constants 129
Result Codes 132

Index 143

Figures, Tables, and Listings

Chapter 1 About Directory Services 13

Figure 1-1 Flow of a directory services request 13
Figure 1-2 Sample directory service 15

Chapter 2 Using Directory Services 21

Listing 2-1 Opening a directory node 21
Listing 2-2 Finding a directory node 23
Listing 2-3 Listing directory nodes 25
Listing 2-4 Clear text authentication 28
Listing 2-5 APOP authentication 31
Listing 2-6 Creating a record 37
Listing 2-7 Listing records in a directory node 38
Listing 2-8 Getting record Information 41
Listing 2-9 Setting the name of a record 42
Listing 2-10 Setting a record's type 44
Listing 2-11 Deleting a record 45

About This Manual

This manual describes the Directory Services API for Mac OS 9 and Mac OS X. The Directory Services API provides an abstraction layer that isolates clients of the API from the actual implementation of a directory system.

Note
The information presented in this manual is preliminary and subject to change. ◆

Conventions Used in This Manual

The Courier font is used to indicate text that you type or see displayed. 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 may be of interest to developers who use the Directory Services API:

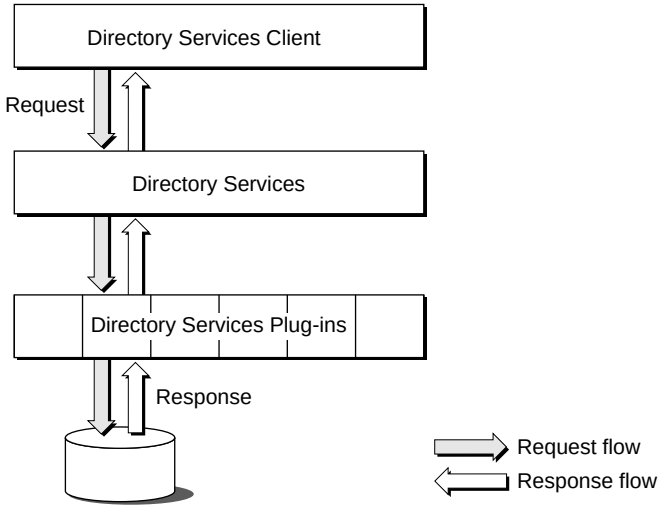
- *TBD a document for writing Directory Services plug-ins.*

About Directory Services

Directory Services provides an abstraction layer designed to isolate clients of the Directory Services API from the actual implementation of a directory system.

When client applications need directory services, they call the appropriate Directory Services function, which passes the request to the plug-in for that directory service. The plug-in obtains the requested information from its directory service and returns the information to Directory Services, which delivers the information to the client application that requested the information.

Figure 1-1 Flow of a directory services request



Note
This document describes an early version of the Directory Services API for client applications. A separate document will describe the Directory Services plug-in API. ◆

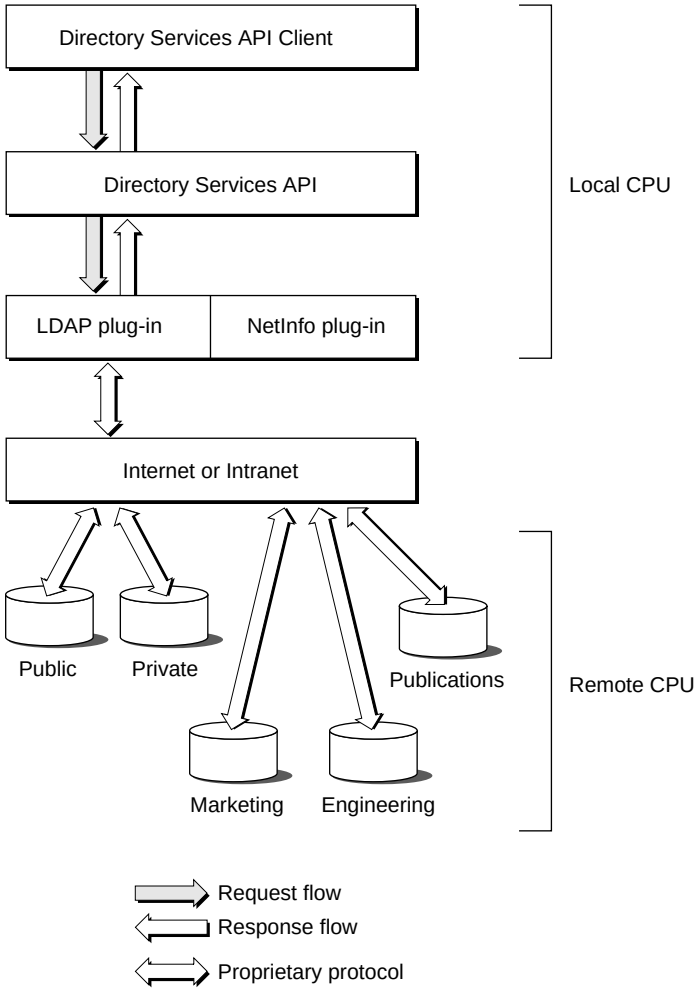
The Directory Services API identifies the basic features that are common to many directory services and provides the minimum set of functions necessary to support the development of high-quality applications that can work with a wide range of dissimilar directory systems.

The Directory Services API is built around the concept of a directory service that is a collection directory nodes. The following rules govern the functionality of directory nodes.

- A **directory node** is either the root of a directory or a child of another directory node.
- Each directory node is a collection of records.
- A record can belong only to one directory node.
- A record has at least one name that is unique within its directory node.
- A record has a type and can be of no more than one type. Examples of record types include user records and group records.
- Directory nodes and records contains any number of **attributes**.
- An attribute contains any number of attribute values.
- **Attribute values** are arbitrary data whose structure is unknown to the Directory Services API. Clients of the Directory Services API are responsible for interpreting the value of any particular attribute. All configuration and discovery of information in the directory system can be accomplished by requesting an attribute value.

Figure 1-2 shows the Directory Services API and two Directory Services plug-ins: LDAP and NetInfo.

Figure 1-2 Sample directory service



Given the directory services shown in Figure 1-2, `dsGetDirNodeList` may return the following list of nodes:

```
NetInfo:Apple Marketing
NetInfo:Apple Engineering
NetInfo:Apple Publications
```

```
Apple LDAP:Public
Apple LDAP:Private
```

The first item in each node is a node prefix. Its value is set when the plug-in is configured.

It is important to note that the plug-in is not required to return information that conforms exactly to the information stored in the directory service. The plug-in can generate information “on the fly” or it may choose to not return information about certain nodes. The plug-in’s behavior in this respect is configurable.

Attributes

Clients of the Directory Services API are responsible for interpreting the value of any particular attribute. All configuration and discovery of information in the directory system can be accomplished by requesting an attribute value.

There are two attribute classes: required attributes and optional attributes. Each record type is required to have certain attributes. Apple Computer will define well-known record types and their required attributes. The well-known record types will include but are not limited to user records, group records, machine records, and printer records.

Providers of services, such as AFP servers, will establish their own well-known record types and publish information about them. Developers are encouraged to use well-known record types whenever possible.

Authentication and Authorization

It is important to distinguish the difference between authentication and authorization:

- **authentication.** A process that uses a piece of information provided by the user (typically a password) to verify the identity of that user.
- **authorization.** The determination of whether a user has permission to access a particular set of information.

Directory Services allows a Directory Service client to use any method to authenticate a user. Directory Services does not provide any facility for

determining whether a user is authorized to access any particular set of information. Moreover, Directory Services does not provide an authorization model. Instead, Directory Services clients are responsible for granting or denying a user access to a particular set of information based on the user's authenticated identity.

When developing an authorization model, Directory Services clients must consider the following:

- what authorization information to store
- where and how to store authorization information
- which applications can see, create, or modify authorization information
- who is authorized to see and change authorization information

Many directory systems store authorization information in the directory system itself. These directory systems use the identity that is currently being used to access the directory system to determine whether to grant access to this information.

Other directory systems store authorization information outside of the directory system. By providing an interface between clients of directory services and the directory services themselves, authorization information that is stored outside of the directory system can be shared. For example, you could design a system that controls authorization based on a common token (such as a user entry in a common directory) so that when an administrator creates, deletes, or modifies a token, all services use that same token for authorization.

With this distinction in mind, the Directory Services function `dsDoDirNodeAuth` has a parameter that indicates to a plug-in whether the proof of identity process is to be used to establish access to the foreign directory or whether the proof of identity process is only a method for checking identity and the foreign directory is being used to determine authorization.

Here are some ways that could be used to establish an identity that is authorized to access a foreign directory system:

- Have the Directory Services client use a preference to establish a "configuration" identity that can access a given directory.
- Configure the Directory Services plug-in with identity information.

It will be necessary for the administrator of the foreign directory to set up, provide, or configure an identity with sufficient access so that a service or

plug-in can access or modify all of the necessary information in the foreign directory system.

Directory Native Authentication

Directory Services supports a mechanism that frees Directory Services clients from having to provide specific information about a particular authentication method. This mechanism is called **directory native authentication**.

When using directory native authentication to authenticate a user to a directory node, the Directory Services client passes to the Directory Services plug-in the user's name, password, and an optional specification that clear text is not an acceptable authentication method.

Upon receipt of the authentication request, the Directory Services plug-in determines the appropriate authentication method based on its configuration (if the plug-in is configurable) or on authentication methods the plug-in has been coded to handle. When the authentication is successful, the Directory Services client receives the authentication type that the plug-in used.

When clear text is the only available authentication method, the plug-in would deny the authentication if the Directory Services client specifies that clear text authentication is unacceptable.

For an example of directory native authentication, see "Native Authentication" (page 31).

Directory Services Metanode

To navigate any directory system, some Directory Service clients may need a place from which to start that is directory-system independent. The Directory Services metanode provides that starting place.

The Directory Services metanode is a well-known node that is easy to locate and is guaranteed to be available.

When a Directory Services client uses the metanode to search for records, the client can request an alias of a fully qualified path to records that match the search criteria.

Searches using the metanode are conducted according to search rules configured in the Directory Services control panel.

Using Directory Services

This chapter presents sample code for performing common Directory Services operations.

Manipulating Directory Nodes

The sample code in this section covers the following topics:

- "Opening a Directory Node" (page 19)
- "Finding a Directory Node" (page 21)
- "Listing Directory Nodes" (page 23)
- "Authenticating Users to Directory Nodes" (page 26)

Note
These examples are designed for NetInfo directories and may not exactly match the Directory Services header file. For the most up-to-date information, contact your developer representative. ◆

Opening a Directory Node

The sample code in this section demonstrates how to open a directory node.

Listing 2-1 Opening a directory node

```
tDirReference dirReference = 0;
tDirReference gDirRef = 0;
```

```
main ( )
{
    tDirNodeReference nodeRef= 0;
    ...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            dsCloseDirNode( nodeRef );
        }
    }
...
}

sInt32 MyOpenDirNode ( tDirNodeReference *outNodeRef )
{
    volatile sInt32 status = eDSNoErr;
    char nodeName[ 256 ] = "\0";
    tDataListPtrnodePath = nil;
    printf( "Open Directroy Node : " );
    scanf( "%s", nodeName );
    printf( "Opening: %s.\n", nodeName );

    nodePath = dsBuildFromPath( gDirRef, nodeName, "/" );
    if ( nodePath != nil )
    {
        status = dsOpenDirNode( gDirRef, nodePath, outNodeRef );
        if ( status == eDSNoErr )
        {
            printf( "Open succeded.  Node Reference = %lu\n", outNodeRef );
        }
        else
        {
            printf( "Open node faild. Err = %ld\n", status );
        }
    }
    return( status );
} // MyOpenDirNode
```

Finding a Directory Node

The sample code in this section demonstrates how to find a specific directory node.

Listing 2-2 Finding a directory node

```
tDirReference gDirRef = 0;

main ( )
{
    tDirNodeReference nodeRef = 0;
...
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        ListNodes();
    }
...

void ListNodes ( void )
{
    Boolean done = false;
    long dirStatus = eDSNoErr;
    unsigned longindex = 0;
    unsigned longnodeCount = 0; //Total number of nodes
    unsigned longbufferCount = 0; //Number of nodes in the buffer
    tDataBufferPtr dataBuffer = nil;
    tDataListPtr nodeName = nil;
    tContextDatacontext = nil;

    dirStatus = dsGetDirNodeCount( gDirRef, &nodeCount );
    printf( "Directory node count is: %lu\n", nodeCount );

    if ( (dirStatus == eDSNoErr) && (nodeCount != 0) )
    {
        //Allocate a 32k buffer.
        dataBuffer = dsDataBufferAllocate( gDirRef, 32 * 1024 );
```

```
        if ( dataBuffer != nil )
        {
            while ( (dirStatus == eDSNoErr) && (done == false) )
            {
                dirStatus = dsGetDirNodeList( gDirRef, dataBuffer, &bufferCount,
                                &context );
                if ( dirStatus == eDSNoErr )
                {
                    for ( index = 0; index < bufferCount; index++ )
                    {
                        nodeName = dsDataListAllocate( gDirRef );
                        if ( nodeName != nil )
                        {
                            dirStatus = dsGetDirNodeName( gDirRef, dataBuffer, index,
                                        &nodeName );
                            if ( dirStatus == eDSNoErr )
                            {
                                printf( "#%4ld ", index );
                                PrintNodeName( nodeName );

                                //Deallocate the nodes.
                                dirStatus = dsDataListDeAllocate( gDirRef, nodeName,
                                                false );
                                nodeName = nil;
                            }
                            else
                            {
                                printf("dsGetDirNodeName error = %ld\n", dirStatus );
                            }
                        }
                    }
                }
                done = (context == nil);
            }
        }
    }
} // ListNodes
```

```
void PrintNodeName ( tDataListPtr inNode )
{
    uInt32 index = 0;
    uInt32 count = 0;
    tDataBufferPriv *pDataNode = nil;

    count = dsDataListGetNodeCount( inNode );

    for ( index = 1; index <= count; index++ )
    {
        pDataNode = nil;
        dsDataListGetNode( inNode, index, (tDataNodePtr *)&pDataNode );
        if ( pDataNode != nil )
        {
            printf( "/%s", pDataNode->fBufferData );
        }
    }

    printf( "\n" );

} // PrintNodeName
```

Listing Directory Nodes

The sample code in this section demonstrates how to display a list of all of the directory nodes.

Listing 2-3 Listing directory nodes

```
tDirReference gDirRef = 0;

main ( )
{
    tDirNodeReference nodeRef= 0;
...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
```

```
        ListNodes();
    }
...

void ListNodes ( void )

{
    Boolean done = false;
    long dirStatus = eDSNoErr;
    unsigned long index = 0;
    unsigned long nodeCount = 0;//total number of nodes
    unsigned long bufferCount = 0;//number of nodes in the buffer
    tDataBufferPtr dataBuffer = nil;
    tDataListPtr nodeName = nil;
    tContextData context = nil;

    dirStatus = dsGetDirNodeCount( gDirRef, &nodeCount );

    printf( "Directory node count is: %lu\n", nodeCount );

    if ( (dirStatus == eDSNoErr) && (nodeCount != 0) )
    {
        //Allocate a 32k buffer.
        dataBuffer = dsDataBufferAllocate( gDirRef, 32 * 1024 );
        if ( dataBuffer != nil )
        {
            while ( (dirStatus == eDSNoErr) && (done == false) )
            {
                dirStatus = dsGetDirNodeList( gDirRef, dataBuffer,
                                &bufferCount, &context );
                if ( dirStatus == eDSNoErr )
                {
                    for ( index = 1; index < bufferCount; index++ )
                    {
                        nodeName = dsDataListAllocate( gDirRef );
                        if ( nodeName != nil )
                        {
                            dirStatus = dsGetDirNodeName( gDirRef,
                                dataBuffer, index, &nodeName );
                            if ( dirStatus == eDSNoErr )
```

```
                            {
                                printf( "#%4ld ", index );
                                PrintNodeName( nodeName );

                                //Deallocate the nodes.
                                dirStatus = dsDataListDeAllocate(
                                    gDirRef, nodeName, false );
                                nodeName = nil;
                            }
                            else
                            {
                                printf("dsGetDirNodeName error = %ld\n",
                                    dirStatus );
                            }
                        }
                    }
                }
                done = (context == nil);
            }
        }
    }
} // ListNodes

void PrintNodeName ( tDataListPtr inNode )
{
    uInt32 index = 0;
    uInt32 count = 0;
    tDataBufferPriv *pDataNode = nil;

    count = dsDataListGetNodeCount( inNode );

    for ( index = 1; index <= count; index++ )
    {
        pDataNode = nil;
        dsDataListGetNode( inNode, index, (tDataNodePtr *)&pDataNode );
        if ( pDataNode != nil )
        {
            printf( "/%s", pDataNode->fBufferData );
        }
    }
```

```
    printf( "\n" );
} // PrintNodeName
```

Authenticating Users to Directory Nodes

To authenticate itself to the Directory Services for the purposes of reading, writing, or making changes to a directory node, a Directory Services client application calls `dsDoDirNodeAuth`. The `dsdoDirNodeAuth` function handles authentication methods that use one or more steps to complete the authentication process.

To determine the authentication methods that a directory node supports, call `dsGetDirNodeInfo` and request the `kDSNAttrAuthMethod` attribute for that directory node.

Clear Text Authentication

This section demonstrates the use of the clear text authentication method to authenticate a user to an opened directory node.

Listing 2-4 Clear text authentication

```
Boolean DoClearTextAuthentication ( const tDirReference inDirRef,
                                    const tDirNodeReference inDirNodeRef,
                                    const char *inUserName, // Name to authenticate
                                    const char *inUserPassword, // Clear text password
                                    const long inUserID ) // User's record ID
{

    /*
        Clear Text is a one-step authentication scheme.
        Step 1
            Send: <length><recordname>
                  <length><cleartextpassword>
                  <length><recordID>
                  <length><recordtype>
             Receive: auth-credential or nothing.
    */
```

```
tDataNodePtr anAuthType2Use = NULL;
tDataBufferPtr anAuthDataBuf = NULL;
tDataBufferPtr aAuthRespBuf = NULL;
tDirStatus aDirErr = eDSNoErr;
tContextData aContinueData = NULL;
long aDataBufSize = 0;
long aTempLength  = 0;
long aCurLength = 0;
Boolean aResult = false;

// Specify the type of authentication.
anAuthType2Use = dsDataNodeAllocate(inDirRef, kDSStdAuthClearText);

aDataBufSize += sizeof(long) + ::strlen(inUserName);
aDataBufSize += sizeof(long) + ::strlen(inUserPassword);
aDataBufSize += sizeof(long) + sizeof(long);
aDataBufSize += sizeof(long) + ::strlen(kDSStdRecordTypeUsers);
anAuthDataBuf = dsDataBufferAllocate(inDirRef, aDataBufSize);
aAuthRespBuf  = dsDataBufferAllocate(inDirRef, 512);// Enough for the response.

// Put all of the authentication arguments into the databuffer.
aTempLength = ::strlen(inUserName);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
        aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserName, aTempLength);
aCurLength += aTempLength;
aTempLength = ::strlen(inUserPassword);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserPassword, aTempLength);
aCurLength += aTempLength;

aTempLength = sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &inUserID, aTempLength);
aCurLength += aTempLength;

aTempLength = ::strlen(kdsStandardUserRecord);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
```

```
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), kDSStdRecordTypeUsers,
                aTempLength);
    aCurLength += aTempLength;

    aDirErr = dsDoDirNodeAuth(inDirNodeRef, anAuthType2Use, true, anAuthDataBuf,
                aAuthRespBuf, &aContinueData);
    switch(aDirErr)
    {
        case eDSNoErr:
            aResult = true;
            break;
        default:
            // If an error occurs, assume the name or password is bad.
            aResult = false;
            break;
    }

    // Clean up allocations that aren't needed anymore.
    aDirErr = dsDataBufferDeAllocate(inDirRef, anAuthDataBuf);
    anAuthDataBuf = NULL;

    // Don't need the response buffer.
    aDirErr = dsDataBufferDeAllocate(inDirRef, aAuthRespBuf);
    aAuthRespBuf = NULL;

    // No need to keep the authentication type value. Build a new one if needed later.
    aDirErr = dsDataNodeDeAllocate(inDirRef, anAuthType2Use);

    anAuthType2Use = NULL;

    // Return the result of the authentication.
    return (aResult);
}
```

APOP Authentication

The sample code in this section demonstrates the Authentication Post Office Protocol (APOP) authentication method.

Listing 2-5 APOP authentication

```
Boolean DoSampleAPOP_Auth ( const tDirReference inDirRef,
                            const tDirNodeReference inDirNodeRef,
                            // inAPOPDigest is the APOP seed value used in the MD5 hash
                            const char *inAPOPDigest,
                            // Name of the user we're attempting to authenticate
                            const char *inUserName,
                            // The APOP result of MD5(inAPOPDigest, ClearTextPassword)
                            const char *inUserPassword,
                            // The numeric value of the user's record ID.
                            const long inUserID )

{
    /*
        APOP is a one step auth scheme. It runs like this.

        Step 1
            Send:
            <length><recordname>
            <length><APOPPasswordHash>
            <length><recordID>
            <length><recordtype>
            <length><APOP Digest>
         Receive: auth-credential or nothing...
    */

    tDataNodePtr anAuthType2Use = NULL;
    tDataBufferPtr anAuthDataBuf = NULL;
    tDataBufferPtr aAuthRespBuf = NULL;
    tDirStatus aDirErr = eDSNoErr;
    tContextData aContinueData = NULL;
    long aDataBufSize = 0;
    long aTempLength = 0;
    long aCurLength = 0;
    Boolean aResult = false;

    // First, "specify" the type of authentication we wish to use
    // See header file for actual constant.
    anAuthType2Use = dsDataNodeAllocate(inDirRef, kDSStdAuthAPOP);
```

```
aDataBufSize += sizeof(long) + ::strlen(inUserName);
aDataBufSize += sizeof(long) + ::strlen(inUserPassword);
aDataBufSize += sizeof(long) + sizeof(long);
aDataBufSize += sizeof(long) + ::strlen(kDSStdRecordTypeUsers);
aDataBufSize += sizeof(long) + ::strlen(inAPOPDigest);
anAuthDataBuf = dsDataBufferAllocate(inDirRef, aDataBufSize);
aAuthRespBuf  = dsDataBufferAllocate(inDirRef, 512);// Enough for the response.

// Put all of the authentication arguments into the databuffer.
aTempLength = ::strlen(inUserName);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserName, aTempLength);
aCurLength += aTempLength;

aTempLength = ::strlen(inUserPassword);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserPassword, aTempLength);
aCurLength += aTempLength;

aTempLength = sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &inUserID, aTempLength);
aCurLength += aTempLength;

aTempLength = ::strlen(kdsStandardUserRecord);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), kDSStdRecordTypeUsers,
    aTempLength);
aCurLength += aTempLength;

aTempLength = ::strlen(inAPOPDigest);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength, sizeof(long));
aCurLength += sizeof(long);
::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inAPOPDigest, aTempLength);
aCurLength += aTempLength;
```

```
    aDirErr = dsDoDirNodeAuth(inDirNodeRef, anAuthType2Use, true, anAuthDataBuf,
            aAuthRespBuf, &aContinueData);

    switch(aDirErr)
    {
        case eDSNoErr:
            aResult = true;
            break;

        default:
            // Any other error, assume the name/password is bad.
            aResult = false;

            break;
    }

    // Clean up allocations

    aDirErr = dsDataBufferDeAllocate(inDirRef, anAuthDataBuf);
    anAuthDataBuf = NULL;

    // Don't need to keep the response,
    aDirErr = dsDataBufferDeAllocate(inDirRef, aAuthRespBuf);
    aAuthRespBuf = NULL;


    // Don't need to keep the authentication type value. Build a new one if needed
later.
    aDirErr = dsDataNodeDeAllocate(inDirRef, anAuthType2Use);
    anAuthType2Use = NULL;

    // Return the result of the authentication
    return (aResult);
}
```

Native Authentication

The sample code in this section demonstrates directory native authentication.

```
Boolean DoNodeNativeAuthentication ( const tDirReference inDirRef,
                                     const tDirNodeReference inDirNodeRef,
                                     // Name of the user to authenticate
                                     const char *inUserName,
                                     // Clear text password
                                     const char *inUserPassword,
                                     // The user's record ID.
                                     const long inUserID )

{
    /*
        Native authentication is a one step authentication scheme.
        It runs like this.

        Step 1

            Send: <length><recordname>
                  <length><cleartextpassword>
                  <length><recordID>
                  <length><recordtype>
             Receive: auth-credential or nothing.
    */

    tDataNodePtr anAuthType2Use = NULL;
    tDataBufferPtr anAuthDataBuf = NULL;
    tDataBufferPtr aAuthRespBuf = NULL;
    tDirStatusaDirErr = eDSNoErr;
    tContextData aContinueData = NULL;
    long aDataBufSize = 0;
    long aTempLength = 0;
    long aCurLength = 0;
    Boolean aResult = false;

    // First, specify the type of authentication.
    anAuthType2Use = dsDataNodeAllocate(inDirRef, kDSStdAuthNodeNativeClearTextOK);

    // The following is an optional method of authentication, that allows
    // the plug-in to choose the authentication method, but the client
    // can "restrict" the authentication request to be "secure" and not
    // use clear text, both authentication methods take the same buffer
    // arguments.
```

```
    /*  anAuthType2Use = dsDataNodeAllocate(inDirRef, kDSStdAuthNodeNativeNoClearText);
*/

    aDataBufSize += sizeof(long) + ::strlen(inUserName);
    aDataBufSize += sizeof(long) + ::strlen(inUserPassword);
    aDataBufSize += sizeof(long) + sizeof(long);
    aDataBufSize += sizeof(long) + ::strlen(kDSStdRecordTypeUsers);
    anAuthDataBuf = dsDataBufferAllocate(inDirRef, aDataBufSize);
    aAuthRespBuf  = dsDataBufferAllocate(inDirRef, 512); // Enough for the response.

    // Put all of the authentication arguments into the databuffer.
    aTempLength = ::strlen(inUserName);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
        sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserName,
            aTempLength);
    aCurLength += aTempLength;
    aTempLength = ::strlen(inUserPassword);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
            sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserPassword,
            aTempLength);
    aCurLength += aTempLength;
    aTempLength = sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
            sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &inUserID,
            aTempLength);
    aCurLength += aTempLength;
    aTempLength = ::strlen(kdsStandardUserRecord);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
            sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]),
            kDSStdRecordTypeUsers, aTempLength);
    aCurLength += aTempLength;
```

```
    aDirErr = dsDoDirNodeAuth(inDirNodeRef, anAuthType2Use, true,
            anAuthDataBuf, aAuthRespBuf, &aContinueData);

    switch(aDirErr)
    {
        case eDSNoErr:
            aResult = true;
            break;

        default:
            // If any other error, assume the name or password is bad.
            aResult = false;
            break;
    }

    // Clean up allocations.
    aDirErr = dsDataBufferDeAllocate(inDirRef, anAuthDataBuf);
    anAuthDataBuf = NULL;

    // Don't need to keep the response.
    aDirErr = dsDataBufferDeAllocate(inDirRef, aAuthRespBuf);
    aAuthRespBuf = NULL;

    // Don't need the authentication type value. Build a new one if needed later.
    aDirErr = dsDataNodeDeAllocate(inDirRef, anAuthType2Use);
    anAuthType2Use = NULL;

    // Return the result of the authentication
    return (aResult);
}
```

Manipulating Records

The sample code in this section covers the following topics:

- “Creating a Record” (page 35)
- “Getting a List of Records” (page 36)

■ “Obtaining Information About a Record” (page 38)
■ “Setting the Name of a Record” (page 40)
■ “Setting the Type of a Record” (page 41)
■ “Deleting a Record” (page 43)

Creating a Record

The sample code in this section demonstrates how to create a record.

Listing 2-6 Creating a record

```
tDirReference gDirRef = 0;

main ( )
{
    tDirNodeReferencenodeRef = 0;
...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef,
                        /* <Directory Node Path List> */ );
        if ( dirStatus == eDSNoErr )
        {
            CreateRecord(nodeRef);
            dsCloseDirNode( nodeRef );
        }
    }
...

void CreateRecord ( const tDirNodeReference inDirNodeRef )
{
    volatile sInt32 status = eDSNoErr;
    tDataNodePtr recName = nil;
    tDataNodePtr recType = nil;
    tDataNodePtr attrName = nil;
```

```
    tDataNodePtr attrValue = nil;
    tRecordReferencerecRef = 0;
    tAccessControlEntry *pACL= nil;

    recName = dsDataNodeAllocateString( gDirRef, "NewUserRecordName" );
    if ( recName != nil )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers );
        if ( recType != nil )
        {
            status = dsCreateRecordAndOpen( inDirNodeRef, recType, recName, &recRef );
            if ( status == eDSNoErr )
            {
                attrName = dsDataNodeAllocateString(gDirRef, "dsAttrTypeNative:Real
Name" );
                if ( attrName != nil )
                {
                    attrValue = dsDataNodeAllocateString( gDirRef, "User Record's
Display Name" );
                    if ( attrValue != nil )
                    {
                        pACL = new tAccessControlEntry;
                        status = dsAddAttribute(recRef, attrName, pACL, attrValue );

                        status = dsDataNodeDeAllocate( gDirRef, attrValue );
                        delete pACL;
                    }
                    status = dsDataNodeDeAllocate( gDirRef, attrName );
                }
            }
            status = dsDataNodeDeAllocate( gDirRef, recType );
        }
        status = dsDataNodeDeAllocate( gDirRef, recName );
    }
} // CreateRecord
```

Getting a List of Records

The sample code in this section demonstrates how to list the records in a directory node.

Listing 2-7 Listing records in a directory node

```
tDirReference gDirRef = 0;

main ( )
{
    tDirNodeReferencenodeRef= 0;

...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef, /* <Directory Node Path> */ );
        if ( dirStatus == eDSNoErr )
        {
            GetRecordList(nodeRef);

            dsCloseDirNode( nodeRef );
        }
    }
...

sInt32 GetRecordList ( const tDirNodeReference inDirNodeRef )
{
    uInt32 i = 0;
    uInt32 j = 0;
    uInt32 k = 0;
    sInt32 status = eDSNoErr;
    uInt32 recCount = 0;
    tDataBufferPtr dataBuff = nil;
    tContextData context = nil;
    tAttributeListRef attrListRef = 0;
    tAttributeValueListRef valueRef = 0;
    tRecordEntry *pRecEntry = nil;
    tAttributeEntry *pAttrEntry = nil;
    tAttributeValueEntry *pValueEntry = nil;
    tDataList recNames;
    tDataList recTypes;
    tDataList attrTypes;
```

```
    dataBuff = dsDataBufferAllocate( gDirRef, 2 * 1024 ); // allocate a 2k buffer
    if ( dataBuff != nil )
    {
        // Status should be checked after each Directory API call. For readability,
        // this code omits checking status after each call.
        status = dsBuildListFromStringsAlloc ( &recNames, kDSRecordsAll, nil );
        status = dsBuildListFromStringsAlloc ( &recTypes, kDSStdRecordTypeUsers, nil );
        status = dsBuildListFromStringsAlloc ( &attrTypes, kDSAttributessAll, nil );
        do
        {
            status = dsGetRecordList( inDirNodeRef, dataBuff, &recNames, eDSExact,
                        &recTypes, &attrTypes, false, &recCount, &context );
            for ( i = 1; i <= recCount; i++ )
            {
                status = dsGetRecordEntry( nodeRef, dataBuff, i, &attrListRef,
                            &pRecEntry );
                for ( j = 1; j <= pRecEntry->fRecordAttributeCount; j++ )
                {
                    status = dsGetAttributeEntry( nodeRef, dataBuff, attrListRef, j,
                                &valueRef, &pAttrEntry );
                    for ( k = 1; k <= pAttrEntry->fAttributeValueCount; k++ )
                    {
                        status = dsGetAttributeValue( nodeRef, dataBuff, k, valueRef,
                                    &pValueEntry );
                        printf( "%s\t- %X\n",
                                    pValueEntry->fAttributeValueData.fBufferData,
                                    pValueEntry->fAttributeValueID );

                        // Whether to deallocate pValueEntry, pAttrEntry, and
                        // pRecEntry is TBD.
            }
        } while (context != NULL);// Loop until all of the data has been obtained.
        // Deallocate recNames, recTypes, attrTypes, dataBuff.
    }
} // GetRecordList
```

Obtaining Information About a Record

The sample code in this section demonstrates how to get information about a record.

Listing 2-8 Getting record Information

```
tDirReferencegDirRef = 0;

main ( )
{
    tDirNodeReference nodeRef = 0;

...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            GetRecInfo(nodeRef);

            dsCloseDirNode( nodeRef );
        }
    }

...

void GetRecInfo ( const  tDirNodeReference inDirNodeRef )
{
    volatile sInt32 status= eDSNoErr;
    tRecordReference recRef = 0;
    tAttributeEntryPtr pAttrInfo = nil;
    tDataNodePtr recName= nil;
    tDataNodePtr recType= nil;
    tDataNodePtr attrType = nil;

    recName = dsDataNodeAllocateString( gDirRef, "appletalk" );
    if ( recName != nil )
    {
        recType = dsDataNodeAllocateString( gDirRef, "dsRecTypeNative:localconfig" );
        if ( recType != nil )
        {
            status = dsOpenRecord( inDirNodeRef, recType, recName,
                    &recRef );
```

```
            if ( status == noErr )
            {
                attrType = dsDataNodeAllocateString(gDirRef,
"dsAttrTypeNative:atalkname" );
                if ( attrType != nil )
                {
                    status = dsGetRecordAttributeInfo(recRef, attrType, &pAttrInfo );
                }
            }
        }

        // Deallocate pAttrInfo, attrType, recName, and recType.
    }
} // GetRecInfo
```

Setting the Name of a Record

The sample code in this section demonstrates how to set the name of a record.

Listing 2-9 Setting the name of a record

```
tDirReferencegDirRef    = 0;

main ( )
{
    tDirNodeReferencenodeRef= 0;

...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            SetRecordName();
            dsCloseDirNode( nodeRef );
        }
    }
```

```
...
}

void SetRecordName ( void )
{
    volatile sInt32status = eDSNoErr;
    tDirNodeReference nodeRef = 0;
    tRecordReference recRef = 0;
    tDataNodePtr recName = nil;
    tDataNodePtr newRecName = nil;
    tDataNodePtr recType = nil;

    recName = dsDataNodeAllocateString( gDirRef, "MikeD" );
    if ( recName != nil )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers );
        if ( recType != nil )
        {
            status = dsOpenRecord( nodeRef, recType, recName, &recRef );
            if ( status == noErr )
            {
                newRecName = dsDataNodeAllocateString(gDirRef, "Robert Smith" );
                if ( newRecName != nil )
                {
                    status = dsSetRecordName( recRef, newRecName );
                    dsDataNodeDeAllocate( gDirRef, newRecName );
                }
            }
            dsDataNodeDeAllocate( gDirRef, recType );
        }
        dsDataNodeDeAllocate( gDirRef, recName );
    }
}
```

Setting the Type of a Record

The sample code in this section demonstrates how to set the type of a record.

Listing 2-10 Setting a record's type

```
tDirReference gDirRef = 0;
main ( )
{
    tDirNodeReference nodeRef = 0;

...

    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            SetRecordType();
            dsCloseDirNode( nodeRef );
        }
    }
...

}

void SetRecordType ( void )
{
    volatile sInt32status   = eDSNoErr;
    tDirNodeReference nodeRef = 0;
    tRecordReference recRef = 0;
    volatile tDataNodePtr recName = nil;
    volatile tDataNodePtr recType = nil;
    volatile tDataNodePtr newRecType = nil;

    recName = dsDataNodeAllocateString( gDirRef, "MikeD" );
    if ( recName != nil )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers );
        if ( recType != nil )
        {
            status = dsOpenRecord( nodeRef, recType, recName, &recRef );
            if ( status == noErr )
            {
```

```
                newRecType = dsDataNodeAllocateString(gDirRef, "dsRecTypeNative:config"
);
                if ( newRecType != nil )
                {
                    status = dsSetRecordType( recRef, newRecType );
                    dsDataNodeDeAllocate( gDirRef, newRecType );
                }
            }
            dsDataNodeDeAllocate( gDirRef, recType );
        }
        dsDataNodeDeAllocate( gDirRef, recName );
    }
}
```

Deleting a Record

The sample code in this section demonstrates how to delete a record.

Listing 2-11 Deleting a record

```
tDirReference gDirRef = 0;
main ( )
{
    tDirNodeReference nodeRef = 0;
...
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            DeleteRecord();
            dsCloseDirNode( nodeRef );
        }
    }
...
}
```

```
void DeleteRecord ( void )
{
    volatile sInt32 status = eDSNoErr;
    tDirNodeReference nodeRef = 0;
    tRecordReference recRef = 0;
    tDataNodePtr recName = nil;
    tDataNodePtr recType = nil;

    recName = dsDataNodeAllocateString( gDirRef, "MikeD" );
    if ( recName != nil )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers );
        if ( recType != nil )
        {
            status = dsOpenRecord( nodeRef, recType, recName, &recRef );
            if ( status == noErr )
            {
                status = dsDeleteRecord( recRef );
            }
            dsDataNodeDeAllocate( gDirRef, recType );
        }
        dsDataNodeDeAllocate( gDirRef, recName );
    }
}
```

Managing Threads

Mac OS 8.x and Mac OS 9.x applications that use the Directory Services APIs must be Mac OS threads compliant. The Directory Services APIs will use Mac OS threads to block and unblock Directory Services API calls because many of these calls will result in network I/O. The APIs will block the calling thread until the API call completes. If Mac OS 8.x applications do not want their HI to be locked up during Directory Service API calls, they need to call the Directory Services API from a Mac OS thread other than their main event loop thread.

The Directory Services API supports the registration of custom block, unblock and yield routines. For information about these custom thread routines, see “Custom Thread Routines” (page 116).

CHAPTER 3

Directory Services Reference

This chapter describes the functions, structures, and data types that you use in order to use Directory Services in your application.

Directory Services Functions

The Directory Services functions are described in these sections:

- "Managing Directory Services Sessions" (page 45)
- "Managing Directory Nodes" (page 47)
- "Managing Records" (page 59)
- "Directory Services Utility Functions" (page 85)

Managing Directory Services Sessions

Before attempting to call Directory Services functions, you must make sure that Directory Services are installed and that its version is compatible with your application.

- `dsOpenDirService` (page 46) opens a Directory Services session.
- `dsCloseDirService` (page 46) closes a Directory Services session.

dsOpenDirService

Opens a Directory Services session.

```
tDirStatus dsOpenDirService(tDirReference *outDirReference);
```

`outDirReference`
On input, a pointer to a value of type `tDirReference` (page 117). On output, the value pointed to by `outDirReference` identifies this session and is passed as a parameter to many Directory Services functions.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsOpenDirService` function opens a Directory Services session. When `dsOpenDirService` returns, use the value pointed to by `outDirReference` when you call Directory Services functions that require a directory reference parameter.

Note
You can establish multiple Directory Services sessions by calling `dsOpenDirService` multiple times. ◆

dsCloseDirService

Closes a Directory Services session.

```
tDirStatus dsCloseDirService(tDirReference inDirReference);
```

`inDirReference` A value of type `tDirReference` (page 117) obtained by previously calling `dsOpenDirService` (page 46) that identifies the Directory Services session that is to be closed.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsCloseDirService` function closes the Directory Services session represented by `inDirReference`. Any data values, such as data lists, data nodes, data buffers, or continuation data, that were created using `inDirReference` become invalid when the session is closed.

Managing Directory Nodes

You use the functions described in this section to manipulate directory nodes.

- `dsCloseDirNode` (page 47) closes a directory node.
- `dsDoDirNodeAuth` (page 50) authenticates a session with a directory node.
- `dsFindDirNodes` (page 52) finds directory nodes that match a pattern.
- `dsGetDirNodeCount` (page 54) obtains the current number of directory nodes associated with a directory service reference.
- `dsGetDirNodeInfo` (page 54) gets information about a node.
- `dsGetDirNodeList` (page 56) obtains a list of directory node entries.
- `dsGetDirNodeName` (page 57) obtains a directory node name.
- `dsOpenDirNode` (page 58) opens a directory node.

dsCloseDirNode

Closes a session with a directory node.

```
tDirStatus dsCloseDirNode(tDirNodeReference inDirNodeReference);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference` obtained by previously calling `dsOpenDirService` (page 46) that identifies the directory node session that is to be closed.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsCloseDirNode` function closes a session with the directory node represented by `inDirNodeReference`.

When the session with the directory node is closed, `inDirNodeReference` becomes invalid and cannot be used with any other Directory Services function that takes a directory node reference as a parameter.

dsDoSetPassword

Sets a password.

```
tDirStatus dsDoSetPassword(
                    tDirNodeReference inDirNodeReference,
                    tDataNodePtr inSetPasswordMethod,
                    Boolean inDirNodeAuthOnlyFlag,
                    tDataBufferPtr inAuthStepData,
                    tDataBufferPtr outAuthStepDataResponse,
                    tContextData *inOutContinueData);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference` obtained by previously calling `dsOpenDirService` (page 46) that identifies the directory node for which the password is to be set.

`inSetPasswordMethod`
On input, a value of type `tDataNodePtr` pointing to `tDataNode` (page 120) containing the authentication method to use. To determine the authentication method that the directory node supports, call `dsGetDirNodeInfo`(page 54) and request the `kDSNAttrAuthMethod` attribute.

`inDirNodeAuthOnlyFlag`
On input, a Boolean value that indicates whether to use the result of setting the password as the basis for granting or denying access to the directory node represented by `inDirNodeReference`. Setting `inDirNodeAuthOnlyFlag` to `TRUE` indicates that you do not want to use the result. Setting `inDirNodeAuthOnlyFlag` to `FALSE` indicates that at the successful completion of the password-setting process, the result should be

used to grant or deny access for subsequent calls to Directory Services for the node represented by `inDirNodeReference`. A file server that wants to authenticate a user but does not want to know how or who is accessing the directory node represented by `inDirNodeReference` should set this parameter to `TRUE`.

`inAuthStepData`

On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure that contains the data for this step in the password-setting process.

`outAuthStepDataResponse`

On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure. On output, this parameter points to the Directory Service plug-in's response. If the password was set successfully, the `tDataBuffer` structure pointed to by `outAuthStepDataResponse` contains an authentication credential that can be used for future authentications for this directory node and other directory nodes in this directory system. If the password was not successfully set, the `tDataBuffer` structured pointed to by this parameter contains a plug-in–defined value. If there are more steps in the password-setting process, this parameter contains a plug-in–defined value that is used in the next step of the password-setting process.

`inOutContinueData`

On input, a pointer to a value of type `tContextData`. On output, if the value pointed to by `inOutContinueData` is `NULL`, there are no more steps in the password-setting process. If `inOutContinueData` is not `NULL` on output, there are more steps to complete. Call `dsDoSetPassword` again and pass to it the value pointed to by `inOutContinueData`. Call `dsReleaseContinueData` (page 108) if the value pointed to by `inOutContinueData` is not `NULL` and you do not complete the password-setting process. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDoSetPassword` function sets a password using a one -step or a multiple-step authentication method.

When the password-setting process completes successfully, the `outAuthStepDataResponse` parameter points to an authentication credential that can be used for future authentications. The current authentication credential can always be obtained by calling `dsGetDirNodeInfo` (page 54) with `kDS1AttrAuthCredential` as one of the requested attributes. Directory nodes that support using a `kDS1AttrAuthCredential` list `DSAuthCredential` as a supported authentication method.

Some directory nodes that support using a previously obtained authentication credential may still fail an authentication for plug-in specific reasons. For example, the plug-in may find the original level of authentication is insufficient for a particular node or configuration or that the credential has expired and become invalid.

dsDoDirNodeAuth

Authenticates a session with a directory node.

```
tDirStatus dsDoDirNodeAuth(
                    tDirNodeReference inDirNodeReference,
                    tDataNodePtr inDirNodeAuthName,
                    Boolean inDirNodeAuthOnlyFlag,
                    tDataBufferPtr inAuthStepData,
                    tDataBufferPtr outAuthStepDataResponse,
                    tContextData *inOutContinueData);
```

`inDirNodeReference`

On input, a value of type `tDirNodeReference` obtained by previously calling `dsOpenDirNode` (page 58) that identifies the directory node session that is to be authenticated.

`inDirNodeAuthName`

On input, a value of type `tDataNodePtr` pointing to a `tDataNode` (page 120) structure containing the name of the authentication

method to use. To determine the authentication method that the directory node supports, call `dsGetDirNodeInfo`(page 54) and request the `kDSNAttrAuthMethod` attribute.

`inDirNodeAuthOnlyFlag`

On input, a Boolean value that indicates whether to use the result of authentication as the basis for granting or denying access to the directory node represented by `inDirNodeReference`. Setting `inDirNodeAuthOnlyFlag` to `TRUE` indicates that you do not want to use the result. Setting `inDirNodeAuthOnlyFlag` to `FALSE` indicates that at the successful completion of the authentication process, the result should be used to grant or deny access for subsequent calls to Directory Services for the node represented by `inDirNodeReference`. A file server that wants to authenticate a user but does not want to know how or who is accessing the directory node represented by `inDirNodeReference` should set this parameter to `TRUE`.

`inAuthStepData`

On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure that contains the data for this step in the authentication process.

`outAuthStepDataResponse`

On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure. On output, the `tDataBuffer` structure contains the plug-in's response. If authentication was successful, the buffer contains an authentication credential that can be used for future authentications for this directory node and other directory nodes supported by the directory service for this node. If the authentication was not successful, the buffer contains a plug-in–defined value. If there are more steps in the authentication process, the buffer contains a plug-in–defined value that is used in the next step of the authentication process.

`inOutContinueData`

On input, a pointer to a value of type `tContextData`. On output, if the value pointed to by `inOutContinueData` is `NULL`, there are no more steps in the authentication process. If `inOutContinueData` is not `NULL` on output, there are more steps to complete. Call `dsDoDirNodeAuth` again and pass to it the value pointed to by

`inOutContinueData`. Call `dsReleaseContinueData` (page 108) if the value pointed to by `inOutContinueData` is not `NULL` and you do not complete the authentication process. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDoDirNodeAuth` function authenticates a session with a directory node using a one-step or a multiple-step authentication method.

When the authentication process completes successfully, the `outAuthStepDataResponse` parameter points to an authentication credential that can be used for future authentications. The current authentication credential can always be obtained by calling `dsGetDirNodeInfo` (page 54) with `kDS1AttrAuthCredential` as one of the requested attributes. Directory nodes that support using a `kDS1AttrAuthCredential` list `DSAuthCredential` as a supported authentication method.

Some directory nodes that support using a previously obtained authentication credential may still fail an authentication for plug-in specific reasons. For example, the plug-in may find the original level of authentication is insufficient for a particular node or configuration or that the credential hasexpired and become invalid.

dsFindDirNodes

Finds the specified directory nodes and puts the results in a data buffer.

```
tDirStatus dsFindDirNodes(
                    tDirReference inDirReference,
                    tDataBufferPtr inOutDataBufferPtr,
                    tDataListPtr inNodeNamePattern,
                    tDirPatternMatch inPatternMatchType,
                    unsigned long *outDirNodeCount,
                    tContextData *inOutContinueData);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inOutDataBufferPtr`
On input, a value of type `tDataBufferPtr` that points to a `tDataBuffer` (page 119) structure in which the results are to be returned. On output, call `dsGetDirNodeName` (page 57) to extract the results from the data buffer pointed to by `inOutDataBufferPtr`.

`inNodeNamePattern`
On input, a value of type `tDataListPtr` pointing to a `tDataList` (page 120) containing the pattern that is to be matched or `NULL` if the local node or the search node is the node that is to be found.

`inPatternMatchType`
On input, a value of type `tDirPatternMatch` that specifies a pattern matching constant, such as the local node, the search node, or a specific pattern. See “Pattern-Matching Constants” (page 127) for defined constants.

`outDirNodeCount` On output, a pointer to a value of type `unsigned long` in which is stored the number of directory node names in the data buffer pointed to by `inOutDataBufferPtr`.

`inOutContinueData`
On input, a pointer to a value of type `tContextData`. If you know that you only want to receive one buffer of response data, set `inOutContinueData` to `NULL` on input. On output, if the value pointed to by `inOutContinueData` is `NULL`, there are no more entries in the buffer. If the value pointed to by `inOutContinueData` is not `NULL` on output, pass the value pointed to by `inOutContinueData` to `dsFindDirNodes` again to get the next entries in the buffer. You must call `dsReleaseContinueData(page 108)` if you don’t want to get the remaining entries. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see “Result Codes” (page 130).

DISCUSSION

The `dsFindDirNodes` function finds all directory nodes in the directory system represented by `inDirReference` that match a particular pattern. Use the `inPatternMatchType` parameter to specify the local node, the search node, or a specific pattern.

dsGetDirNodeCount

Gets the total number of directory nodes for a directory service reference.

```
tDirStatus dsGetDirNodeCount(
                    tDirReference inDirReference,
                    unsigned long *outDirectoryNodeCount);
```

`inDirReference` A value of type `tDirReference` (page 117) obtained by previously calling `dsOpenDirService` (page 46).

`outDirectoryNodeCount`
On output, a pointer to a value of type `unsigned long` containing the total number of directory nodes in the directory represented by `inDirReference`.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetDirNodeCount` function gets the total number of directory nodes for the specified directory service reference.

dsGetDirNodeInfo

Gets information about a directory node andputs it into a data buffer.

```
tDirStatus dsGetDirNodeInfo(
                    tDirNodeReference inDirNodeReference,
                    tDataListPtr inDirNodeInfoTypeList,
```

```
                    tDataBufferPtr inOutDataBuffer,
                    Boolean inAttributeInfoOnly,
                    unsigned long *outAttributeInfoCount,
                    tAttributeListRef *outAttributeListRef,
                    tContextData *inOutContinueData);
```

`inDirNodeReference`
On input, a directory service reference obtained by calling `dsOpenDirService` (page 46).

`inDirNodeInfoTypeList`
On input, a value of type `tDataListPtr` pointing to a `tDataList` (page 120) structure containing the attribute types for which information is being requested.

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure. On output, the `tDataBuffer` structure contains the requested directory node information.

`inAttributeInfoOnly`
On input, a Boolean value set to `TRUE` if the calling application only wants information about attributes. To get the values of the attributes as well as information about the attributes, set `inAttributeInfoOnly` to `FALSE`.

`outAttributeInfoCount`
On input, a pointer to a value of type `unsigned long`. On output, `outAttributeInfoCount` points to the number of attribute types present in the buffer pointed to by `inOutDataBuffer`.

`outAttributeListRef`
On input, a value of type `tAttributeListRef` that represents an `tAttributeEntry` (page 122). On output, you can pass `outAttributeListRef` to `dsGetAttributeEntry` (page 77) to obtain information about an attribute.

`inOutContinueData`
On input, a pointer to a value of type `tContextData`. If you know that you only want to receive one buffer of response data, set `inOutContinueData` to `NULL` on input. On output, if `inOutContinueData` is not `NULL`, pass the pointer to `inOutContinueData` to `dsGetDirNodeInfo` again to get the next entries in the buffer. If on output the value of `inOutContinueData` is `NULL`, there are no more entries in the buffer. Call

dsReleaseContinueData(page 108) if inOutContinueData is not NULL and you don't want to get the remaining entries. Values of type tContextData behave like a Pascal var or a C++ reference.

function result A value of type tDirStatus indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The dsGetDirNodeInfo function obtains information about a directory node, such as the authentication methods it supports, its unique ID, icon information, access controls, the record types the directory node contains, and the directory node's type and signature.

dsGetDirNodeList

Gets a list of directory nodes and puts it in a data buffer.

```
tDirStatus dsGetDirNodeList(
                    tDirReference inDirReference,
                    tDataBufferPtr inOutDataBufferPtr,
                    unsigned long *outDirNodeCount,
                    tContextData *inOutContinueData);
```

inDirReference
On input, a value of type tDirReference (page 117) obtained by previously calling dsOpenDirService (page 46).

inOutDataBufferPtr
On input, a value of type tDataBufferPtr pointing to a tDataBuffer (page 119) structure. On output, the tDataBuffer contains the requested list of directory nodes. Call dsGetDirNodeName (page 57) to extract the names of the directory nodes in the list.

outDirNodeCount On output, a pointer to a value of type unsigned long containing the number of directory node names in the tDataBuffer pointed to by inOutDataBufferPtr.

`inOutContinueData`
On input, a pointer to a value of type `tContextData`. If you know that you only want to receive one buffer of response data, set `inOutContinueData` to `NULL` on input. On output, if `inOutContinueData` is not `NULL`, pass the pointer to `inOutContinueData` to `dsGetDirNodeList` again to get the next entries in the buffer. If on output the value of `inOutContinueData` is `NULL`, there are no more entries in the buffer. Call `dsReleaseContinueData(page 108)` if `inOutContinueData` is not `NULL` and you don't want to get the remaining entries. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetDirNodeList` function gets a list of directory nodes and puts it in a data buffer. Call `dsGetDirNodeName` (page 57) to extract the names from the data buffer.

dsGetDirNodeName

Gets the names of directory nodes from a data buffer.

```
tDirStatus dsGetDirNodeName(
                    tDirReference inDirReference,
                    tDataBufferPtr inOutDataBuffer,
                    unsigned long inDirNodeIndex,
                    tDataListPtr *inOutDataList);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by previously calling `dsOpenDirService` (page 46).

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure containing the results of calling `dsGetDirNodeList` (page 56) or `dsFindDirNodes` (page 52).

`inDirNodeIndex` On input, a value of type `unsigned long`. Set `inDirNodeIndex` to 1 to get the first name. Set `inDirNodeIndex` to 2 to get the second name, and so on.

`inOutDataList` On input, a value of type `tDataListPtr` that points to a `tDataList` (page 120) structure. On output, the data list contains the full pathname of the item specified by `inDirNodeIndex` in the buffer pointed to by `inOutDataBuffer`. The calling application is responsible for disposing of the data list by calling `dsDataListDeAllocate` (page 96).

DISCUSSION

The `dsGetDirNodeName` function parses a buffer of directory node names obtained by calling `dsFindDirNodes` (page 52) or `dsGetDirNodeList` (page 56). It returns a `tDataList` structure containing the full path name of the directory node in the buffer pointed to by `inOutDataBuffer`.

dsOpenDirNode

Opens a session with a directory node.

```
tDirStatus dsOpenDirNode(
                    tDirReference inDirReference,
                    tDataListPtr inDirNodeName,
                    tDirNodeReference *outDirNodeReference);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by previously calling `dsOpenDirService` `dsOpenDirService` (page 46).

`inDirNodeName` On input, a value of type `tDataListPtr` that points to a `tDataList` (page 120) structure containing the name of the directory node to open. You can get the name of the directory node by calling `dsGetDirNodeList` (page 56) or by calling `dsBuildListFromStrings` (page 93) to construct the name yourself.

`outDirNodeReference`
On input, a pointer to a value of type `tDirNodeReference`. On output, the value pointed to by `outDirNodeReference` contains a directory node session reference that can be provided as a

parameter to Directory Services functions that manipulate directory nodes, such as `dsGetDirNodeInfo` (page 54), `dsDoDirNodeAuth` (page 50), `dsGetRecordList` (page 64), `dsGetRecordEntry` (page 63), `dsOpenRecord` (page 67), and `dsGetAttributeEntry` (page 77).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsOpenDirNode` function opens a session with the directory node whose name is specified in the `tDataList` structure pointed to by the `inDirNodeName` parameter. Opening a session with a directory node allows you to perform operations on the opened directory node, such as creating, listing, and removing records.

Managing Records

You use the functions described in this section to manipulate records.

- `dsCloseRecord` (page 60) closes a record.
- `dsCreateRecord` (page 60) creates a record.
- `dsCreateRecordAndOpen` (page 61) creates a record and opens it.
- `dsDeleteRecord` (page 62) deletes a record.
- `dsFlushRecord` (page 63) writes a record.
- `dsGetRecordEntry` (page 63) gets a record from a buffer.
- `dsGetRecordList` (page 64) obtains a list of records.
- `dsGetRecordReferenceInfo` (page 67) obtains the name and type of a record.
- `dsOpenRecord` (page 67) opens a record.
- `dsSetRecordAccess` (page 69) sets the access controls for a record.
- `dsSetRecordFlags` (page 69) sets the flags of a record.
- `dsSetRecordName` (page 70) sets the name of a record.
- `dsSetRecordType` (page 71) sets the type of a record.

dsCloseRecord

Closes an open record.

```
tDirStatus dsCloseRecord(tRecordReference inRecordReference);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that identifies the record that is to be closed.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsCloseRecord` function closes a record that was previously opened by calling `dsOpenRecord` (page 67). Closing the record invalidates the `inRecordReference` parameter so that it cannot be used as a parameter to any other Directory Services function.

dsCreateRecord

Creates a record.

```
tDirStatus dsCreateRecord(
                    tDirNodeReference inDirNodeReference,
                    tDataNodePtr inRecordType,
                    tDataNodePtr inRecordName);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference`, obtained by previously calling `dsOpenDirNode` (page 58), that identifies the directory node in which the record is to be created.

`inRecordType` On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type that is to be assigned to the created record.

`inRecordName` On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the name that is to be assigned to the created record.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsCreateRecord` function creates a record having the name and type specified by the data notes pointed to by the `inRecordType` and `inRecordName` parameters in the directory node represented by `inDirNodeReference`.

To add attributes to the new record, call `dsAddAttribute` (page 72).

The `dsCreateRecord` function does not open the created record. To create a record and open it in one step, call `dsCreateRecordAndOpen` (page 61).

dsCreateRecordAndOpen

Creates a record and opens it.

```
tDirStatus dsCreateRecordAndOpen(
                    tDirNodeReference inDirNodeReference,
                    tDataNodePtr inRecordType,
                    tDataNodePtr inRecordName
                    tRecordReference *outRecordReference);
```

`inDirNodeReference` On input, a value of type `tDirNodeReference`, obtained by calling `dsOpenDirNode` (page 58) that identifies the directory node in which the record is to be created.

`inRecordType` On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type that is to be assigned to the created record.

`inRecordName` On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the name that is to be assigned to the created record.

`outRecordReference`
On input, a value of type `tRecordReference`. On output, `ouRecordReference` represents the record that was created and can be provided as a parameter to Directory Services functions that operate on records.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsCreateRecordAndOpen` function creates a record and opens it. On output the `ouRecordReference` parameter is a reference to the newly created record.

To add attributes to the new record, call `dsAddAttribute` (page 72).

To create a record without opening it, call `dsCreateRecord` (page 60).

dsDeleteRecord

Deletes a record.

```
tDirStatus dsDeleteRecord(tRecordReference inRecordReference);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDeleteRecord` function deletes the specified record. Deleting the record invalidates the `inRecordReference` parameter so that it cannot be used for any other Directory Services functions that take a value of type `tRecordReference` as a parameter.

dsFlushRecord

Writes a record.

```
tDirStatus dsFlushRecord(tRecordReference inRecordReference);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsFlushRecord` function requests the directory system to write the record. The directory system may comply with the request or may choose to ignore it.

The value returned by `dsFlushRecord` does not reflect whether the record was actually written.

dsGetRecordEntry

Gets the next record from a data buffer.

```
tDirStatus dsGetRecordEntry
                    tDirNodeReference inDirNodeReference,
                    tDataBufferPtr inOutDataBuffer,
                    unsigned long inRecordEntryIndex,
                    tAttributeListRef *outAttributeListRef,
                    tRecordEntryPtr *outRecordEntryPtr);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference`, obtained by calling `dsOpenDirNode` (page 58), that identifies the directory node in which the record specified by `inRecordEntryIndex` resides.

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure containing data obtained by previously calling `dsGetRecordList` (page 64).

`inRecordEntryIndex`
On input, a value of type `unsigned long` that specifies the next record to get. Set `inRecordEntryIndex` to 1 to get the first record. Set `inRecordEntryIndex` to 2 to get the second record, and so on.

`outAttributeListRef`
On input, a pointer to a value of type `tAttributeListRef`. On output, you can pass the pointer to `outAttributeListRef` as a parameter when calling `dsGetAttributeEntry` (page 77) to obtain information about an attribute.

`outRecordEntryPtr`
On output, a pointer to a value of type `tRecordEntryPtr` that points to a `tRecordEntry` (page 123) structure containing the specified record.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordEntry` function extracts the specified record from the data buffer pointed to by `inOutDataBuffer` and puts it in the `tRecordEntry` (page 123) structure pointed to by `outRecordEntryPtr`.

This function also uses the `outAttributeListRef` parameter to return an attribute list reference that can be used in subsequent calls to `dsGetAttributeValue` (page 76).

dsGetRecordList

Gets a list of records and puts it in a data buffer.

```
tDirStatus dsGetRecordList(
                    tDirNodeReference inDirNodeReference,
                    tDataBufferPtr inOutDataBuffer,
                    tDataListPtr inRecordNameList,
                    tDirPatternMatch inPatternMatchType,
                    tDataListPtr inRecordTypeList,
                    tDataListPtr inAttributeTypeList,
```

```
                    Boolean inAttributeInfoOnly,
                    unsigned long *outRecordEntryCount,
                    tContextData *inOutContinueData);
```

`inDirNodeReference`
: On input, a directory service reference obtained by calling `dsOpenDirService` (page 46).

`inOutDataBuffer`
: On input, a value of type `tDataBufferPtr` that points to a `tDataBuffer` (page 119) structure into which `dsGetRecordList` places the requested records. On output, parse the records by passing the pointer to `inOutDataBuffer` to `dsGetRecordEntry` (page 63).

`inRecordNameList`
: On input, a value of type `tDataListPtr` pointing to a `tDataList` (page 120) structure that contains a list of record names to be matched. To get all record names, set `inRecordNameList` to `NULL`.

`inPatternMatchType`
: On input, a value of type `tDirPatternMatch` that specifies a pattern that is to be matched. See “Pattern-Matching Constants” (page 127) for defined constants. The `inPatternMatchType` parameter is ignored if the `inRecordNameList` parameter is `NULL`.

`inRecordTypeList`
: On input, a value of type `tDataListPtr` pointing to a data list containing the types of records to get. To get records of all types, set `inRecordTypeList` to `NULL`.

`inAttributeTypeList`
: On input, a value of type `tDataListPtr` pointing to a `tDataList` (page 120) structure that contains the attribute types of the records that are to be obtained. To get records of all attribute types, set `inAttrbuteTypeList` to `NULL`.

`inAttributeInfoOnly`
: On input, a Boolean value set to `TRUE` if the calling application wants information about attributes. To get information about the attributes and the value of the attributes, set `inAttributeInfoOnly` to `FALSE`.

`outRecordEntryCount`

On input, a pointer to a value of type `long`. On output, `outRecordEntryCount` points to the number of recordsin the data buffer pointed to by `inOutDataBuffer`.

`inOutContinueData`

On input, a pointer to a value of type `tContextData`. If you know that you only want to receive one buffer of response data, set `inOutContinueData` to `NULL` on input. On output, if the value pointed to by `inOutContinueData` is `NULL`, there are no more entries in the buffer. If the value pointed to by `inOutContinueData` is not `NULL` on output, pass the value pointed to by `inOutContinueData` to `dsGetRecordList` again to get the next entries in the buffer. You must call `dsReleaseContinueData(page 108)` if you don't want to get the remaining entries. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordList` function obtains a list of records having the specified data types and values. Call `dsGetRecordEntry` (page 63) to parse the records in the list.

If there are too many records to fit in the `tDataBuffer` pointed to by `inOutDataBuffer`, `dsGetRecordList` returns a non-null value in the value pointed to by `inOutContinueData`. To get more records, call `dsGetRecordList` again, passing the pointer to the `inOutContinueData` parameter that was returned by the previous call to `dsGetRecordList`.

If the value pointed to by `inOutContinueData` is not `NULL` and you do not want to get more records, call `dsReleaseContinueData` (page 108) to release the memory associated with `inOutContinueData`.

dsGetRecordReferenceInfo

Gets the name and type of a record.

```
tDirStatus dsGetRecordReferenceInfo(
                    tRecordReference inRecordReference,
                    tRecordEntryPtr *outRecordInfo);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record reference information is to be obtained.

`outRecordInfo` On output, a pointer to a value of type `tRecordEntryPtr` that points to a `tRecordEntry` (page 123) structure containing the record information for the specified record.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordReferenceInfo` function gets information about the record represented by `inRecordReference` and stores it in a `tRecordEntry` structure.

The information includes record flags, access controls, the number of attributes the record has, and the name and type of the record.

dsOpenRecord

Opens a record.

```
tDirStatus dsOpenRecord(
                    tDirNodeReference inDirNodeReference,
                    tDataNodePtr inRecordType,
                    tDataNodePtr inRecordName,
                    tRecordReference *outRecordReference);
```

`inDirNodeReference`
On input, a directory node reference obtained by calling `dsOpenDirNode` (page 58).

`inRecordType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the record to open.

`inRecordName`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the name of the record to open.

`outRecordReference`
On output, a pointer to a value of type `tRecordReference` (page 123) that you can pass to other Directory Services functions that operate on records, such as `dsGetRecordReferenceInfo` (page 67), `dsFlushRecord` (page 63), `dsSetRecordName` (page 70), `dsSetRecordType` (page 71), `dsSetRecordAccess` (page 69), `dsSetRecordFlags` (page 69), and `dsCloseRecord` (page 60).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsOpenRecord` function opens a record and returns in the value pointed to by the `outRecordReference` parameter a record reference that you can use in subsequent calls to Directory Services functions that manipulate records.

A record must be open before you can perform operations on the record, such as setting its access controls, its name and type, and its record flags, adding attributes, setting attribute values, and deleting the record.

To close an open record, call `dsCloseRecord` (page 60).

dsSetRecordAccess

Sets the access controls for a record.

```
tDirStatus dsSetRecordAccess(
                    tRecordReference inRecordReference,
                    tAccessControlEntryPtr inNewRecordAccess);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record whose access controls are to be set.

`inNewRecordAccess`
On input, a value of type `tAccessControlEntryPtr` pointing to a `tAccessControlEntry` (page 121) structure that specifies the access controls that are to be set for the record.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetRecordAccess` function sets a record's access controls.

dsSetRecordFlags

Sets a record's flags.

```
tDirStatus dsSetRecordFlags(
                    tRecordReference inRecordReference,
                    unsigned long inRecordFlags);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record whose flags are to be set.

`inRecordFlags` On input, a value of type `unsigned long` that specifies the flags that are to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetRecordFlags` function sets a record's flags.

dsSetRecordName

Sets the name of a record.

```
tDirStatus dsSetRecordName(
                    tRecordReference inRecordReference,
                    tDataNodePtr inNewRecordName);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record whose name is to be set.

`inNewRecordName`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the name that is to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetRecordName` function sets the name of a record.

dsSetRecordType

Sets the type of a record.

```
tDirStatus dsSetRecordType(
                    tRecordReference inRecordReference,
                    tDataNodePtr inNewRecordType);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record whose type is to be set.

`inNewRecordType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type that is to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetRecordType` function sets the type of a record.

Working with Attributes

You use the functions described in this section to manipulate attributes.

- `dsAddAttribute` (page 72) adds an attribute to a record.
- `dsAddAttributeValue` (page 73) adds data to an attribute value.
- `dsDoAttributeValueSearch` (page 74) conducts a search based on the value of an attribute.
- `dsGetAttributeValue` (page 76) obtains the value of an attribute.
- `dsGetAttributeEntry` (page 77) obtains information about attributes.
- `dsGetRecordAttributeInfo` (page 78) obtains information about a record's attributes by attribute type.

- dsGetRecordAttributeValueByID (page 79) uses an attribute ID to obtain the value of an attribute.
- dsGetRecordAttributeValueByIndex (page 80) uses an index to obtain the value of an attribute.
- dsRemoveAttribute (page 81) removes an attribute from a record.
- dsRemoveAttributeValue (page 82) removes an attribute's value.
- dsSetAttributeFlags (page 83) sets the flags for an attribute.
- dsSetAttributeAccess (page 83) sets the access controls for an attribute.
- dsSetAttributeValue (page 84) sets an attribute's value.

dsAddAttribute

Adds an attribute to a record.

```
tDirStatus dsAddAttribute(
                    tRecordReference inRecordReference,
                    tDataNodePtr inNewAttribute,
                    tAccessControlEntryPtr inNewAttributeAccess,
                    tDataNodePtr inFirstAttributeValue);
```

inRecordReference
: On input, a record reference obtained by previously calling dsOpenRecord (page 67) that represents the record to which an attribute is to be added.

inNewAttribute
: On input, a value of type tDataNodePtr that points to a tDataBuffer (page 119) structure containing the name of the attribute that is to be added.

inNewAttributeAccess
: On input, a value of type tAccessControlEntryPtr that points to a tAccessControlEntry (page 121) structure specifying the access controls for the attribute that is to be added.

`inFirstAttributeValue`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the value of the attribute that is to be added.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsAddAttribute` function adds an attribute to a record.

dsAddAttributeValue

Adds data to an attribute value.

```
tDirStatus dsAddAttributeValue(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    tDataNodePtr inAttributeValue);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) that represents the record having an attribute to which data is to be appended.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attributeto which data is to be appended.

`inAttributeValue`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the attribute value that is to be appended.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsAddAttributeValue` function appends the specified data to the specified attribute's existing data.

To replace the value of an attribute, call `dsSetAttributeValue` (page 84).

dsDoAttributeValueSearch

Searches for attributes by value and puts the results in a data buffer.

```
tDirStatus dsDoAttributeValueSearch(
                    tDirNodeReference inDirNodeReference,
                    tDataBufferPtr inOutDataBuffer,
                    tDataListPtr inRecordTypeList,
                    tDataNodePtr inAttributeType,
                    tDirPatternMatch inPatternMatchType,
                    tDataNodePtr inPattern2Match,
                    unsigned long *outMatchRecordCount,
                    tContextData *inOutContinueData);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference`, obtained by calling `dsOpenDirNode` (page 58), that identifies the directory node that is to be searched.

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure. On output, the `tDataBuffer` structure contains the search results. Call `dsGetRecordEntry` (page 63), `dsGetAttributeEntry` (page 77), and `dsGetAttributeValue` (page 76) to get the records, attributes, and attribute values from the data buffer.

`inRecordTypeList`
On input, a value of type `tDataListPtr` pointing to a `tDataList` (page 120) structure containing record types to search for.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing a list of attribute types to search for.

`inPatternMatchType`
: On input, a value of type `tDirPatternMatch` specifying the pattern type to use. See “Pattern-Matching Constants” (page 127) for possible values.

`inPattern2Match`
: On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the pattern to match.

`outMatchRecordCount`
: On input, a pointer to a value of type `unsigned long`. On output, `outMatchRecordCount` points to the number of records that matched.

`inOutContinueData`
: On input, a pointer to a value of type `tContextData`. If you know that you only want to receive one buffer of response data, set `inOutContinueData` to `NULL` on input. On output, if the value pointed to by `inOutContinueData` is `NULL`, there are no more entries in the buffer. If the value pointed to by `inOutContinueData` is not `NULL` on output, pass the value pointed to by `inOutContinueData` to `dsDoAttributeValueSearch` again to get the next entries in the buffer. Call `dsReleaseContinueData(page 108)` if you don’t want to get the remaining entries. Values of type `tContextData` behave like a Pascal `var` or a C++ reference.

function result
: A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see “Result Codes” (page 130).

DISCUSSION

The `dsDoAttributeValueSearch` function stores in the data buffer pointed to by `inOutDataBuffer` a list of records having attributes whose values match the specified pattern. If there are too many records to fit in a single buffer, `dsDoAttributeValueSearch` returns a non-null value in the value pointed to by `inOutContinueData`. To get more records, call `dsDoAttributeValueSearch` again, passing the pointer to `inOutContinueData` that was returned by the previous call to `dsDoAttributeValueSearch`.

To get a record from the data buffer pointed to by `inOutDataBuffer`, call `dsGetRecordEntry` (page 63). To get information about the record’s attributes, call

`dsGetAttributeEntry` (page 77). To get the value of a record's attribute, call `dsGetAttributeValue` (page 76).

When you no longer need `inOutContinueData`, call `dsReleaseContinueData` (page 108) to release the memory associated with it.

dsGetAttributeValue

Gets the value of an attribute from a data buffer.

```
tDirStatus dsGetAttributeValue(
                    tDirNodeReference inDirNodeReference,
                    tDataBufferPtr inOutDataBuffer,
                    unsigned long inAttributeValueIndex,
                    tAttributeValueListRef inAttributeValueListRef,
                    tAttributeValueEntryPtr *outAttributeValue);
```

`inDirNodeReference`
On input, a directory service reference obtained by calling `dsOpenDirNode` (page 58) that represents the direcotry node for which the search was conducted.

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure that was previously filled in by calling `dsDoAttributeValueSearch` (page 74).

`inAttributeValueIndex`
On input, a value of type `unsigned long`. Set `inAttributeValueIndex` to 1 to get the first attribute value. Set `inAttributeValueIndex` to 2 to get the second attribute value, and so on.

`inAttributeValueListRef`
On input, a value of type `tAttributeValueListRef` that represents a `tAttributeValueEntry` (page 122) structure containing an attribute value ID and the value of the attribute represented by the attribute value ID.

`outAttributeValue`
On output, a pointer to a value of type `tAttributeValueEntryPtr` that points to a `tAttributeValueEntry` (page 122) structure containing the value ID and the value of the attibute.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetAttributeValue` function obtains the value of an attribute from a data buffer previously filled in by calling `dsDoAttributeValueSearch` (page 74) and stores the value in a `tAttributeValueEntry` (page 122) structure allocated by the client.

dsGetAttributeEntry

Gets an attribute entry from a data buffer.

```
tDirStatus dsGetAttributeEntry(
                    tDirNodeReference inDirNodeReference,
                    tDataBufferPtr inOutDataBuffer,
                    tAttributeListRef inAttributeListRef,
                    unsigned long inAttributeInfoIndex,
                    AttributeValueListRef *outAttributeValueListRef,
                    tAttributeEntryPtr *outAttributeInfoPtr);
```

`inDirNodeReference`
On input, a directory service reference obtained by calling `dsOpenDirNode` (page 58) that represents the directory node v for which the search was conducted.

`inOutDataBuffer` On input, a value of type `tDataBufferPtr` pointing to a `tDataBuffer` (page 119) structure filled in by previously calling `dsDoAttributeValueSearch` (page 74).

`inAttributeListRef`
On input, a value of type `tAttributeListRef` obtained by previously calling `dsGetDirNodeInfo` (page 54) or `dsGetRecordEntry` (page 63).

`inAttributeInfoIndex`
On input, a value of type `unsigned long`. Set `inAttributeInfoIndex` to 1 to get the first attribute entry. Set `inAttributeInfoIndex` to 2 to get the second attribute entry, and so on.

`outAttributeValueListRef`
On output, a pointer to a value of type `tAttributeValueListRef`. Pass the pointer to `outAttributeValueListRef` to `dsGetAttributeValue` (page 76) to get the value of the attribute.

`outAttributeInfoPtr`
On output, a pointer to a value of type `tAttributeEntryPtr` that points to a `tAttributeEntry` (page 122) structure in which information about the attribute is returned. The information includes attribute flags, the attribute's access controls, the number of attribute values, the maximum size of the attribute's value, and the attribute's signature.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetAttributeEntry` function uses a data buffer obtained by calling `dsGetRecordList` (page 64) or `dsGetDirNodeInfo` (page 54) and an attribute list reference to get information about attributes (but not the actual data). The function stores the information in the `tAttributeEntry` (page 122) structure pointed to by `outAttributeInfoPtr`.

To get the value of the attribute, call `dsGetAttributeValue` (page 76).

dsGetRecordAttributeInfo

Uses an attribute type to obtain attribute informationß.

```
tDirStatus dsGetRecordAttributeInfo(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    tAttributeEntryPtr *outAttributeInfoPtr);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record for which record information is to be obtained.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the attribute type for which information is to be obtained.

`outAttributeInfoPtr`
On output, a pointer to a value of type `outAttributeEntryPtr` that points to a `tAttributeEntry` (page 122) structure containing the information about the attribute pointed to by `inAttributeType`.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordAttributeInfo` function uses an attribute type to obtain information about an attribute for the record represented by `inRecordReference`. The information consists of the attribute's flags, access controls, value count, data size, maximum value size, and signature.

dsGetRecordAttributeValueByID

Uses an attribute ID to obtain the value of an attribute.

```
tDirStatus dsGetRecordAttributeValueByID(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    unsigned long inValueID,
                    tAttributeValueEntryPtr *outEntryPtr);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record having an attribute whose value is to be obtained.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute whose value is to be obtained.

`inValueID` On input, a value of type `unsigned long` containing the attribute ID of the attribute whose value is to be obtained. Call `dsGetAttributeValue` (page 76) to get the attribute's ID.

`outEntryPtr` On output, a pointer to a value of type `tAttributeValueEntryPtr` that points to a `tAttributeValueEntry` (page 122) structure containing the requested attribute value.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordAttributeValueByID` function uses an attribute ID to obtain the value of an attribute for the record represented by `inRecordReference`.

dsGetRecordAttributeValueByIndex

Uses an index to obtain the value of an attribute.

```
tDirStatus dsGetRecordAttributeValueByIndex(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    unsigned long inValueIndex,
                    tAttributeValueEntryPtr *outEntryPtr;
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record that has an attribute whose value is to be obtained.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute whose value is to be obtained.

`inValueIndex` On input, a value of type `unsigned long` that specifies the index of the attribute value that is to be obtained.

`outEntryPtr` On output, a value of type `tAttributeValueEntryPtr` that points to a `tAttributeValueEntry` (page 122) structure containing the requested attribute value.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetRecordAttributeValueByIndex` function uses an index to obtain the value of an attribute for the record represented by `inRecordReference`.

To determine whether an attribute can have multiple values, call `dsGetAttributeEntry` (page 77) or `dsGetRecordAttributeInfo` (page 78) which returns a value of type `tAttributeEntryPtr` that points to the attribute's value count.

dsRemoveAttribute

Removes an attribute from a record.

```
tDirStatus dsRemoveAttribute(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttribute);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67).

`inAttribute` On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the name of the attribute that is to be removed.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsRemoveAttribute` function removes an attribute from a record.

dsRemoveAttributeValue

Removes an attribute value.

```
tDirStatus dsRemoveAttributeValue(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    unsigned long inAttributeValueID);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record from which an attribute is to be removed.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute that is to be removed.

`inAttributeValueID`
On input, a value of type `unsigned long` that specifies the attribute ID of the attribute whose value is to be removed. Call `dsGetAttributeValue` (page 76) to get the attribute ID.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsRemoveAttributeValue` function removes an attribute value.

dsSetAttributeAccess

Sets the access controls for an attribute.

```
tDirStatus dsSetAttributeAccess(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    tAccessControlEntryPtr inAttributeAccess);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record that has an attribute whose access controls are to be set.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute whose access controls are to be set.

`inAttributeAccess`
On input, a value of type `tAccessControlEntryPtr` that points to a `tAccessControlEntry` (page 121) structure containing the access controls that are to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetAttributeAccess` function sets the access controls for an attribute.

dsSetAttributeFlags

Sets the flags for an attribute.

```
tDirStatus dsSetAttributeFlags(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    unsigned long inAttributeFlags);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record that has an attribute whose flags are to be set.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute whose flags are to be set.

`inAttributeFlags`
On input, a value of type `unsigned long` that specifies the flags that are to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetAttributeFlags` function sets the flags for an attribute.

dsSetAttributeValue

Sets the value of an attribute.

```
tDirStatus dsSetAttributeValue(
                    tRecordReference inRecordReference,
                    tDataNodePtr inAttributeType,
                    tAttributeValueEntryPtr inAttributeValuePtr);
```

`inRecordReference`
On input, a record reference obtained by previously calling `dsOpenRecord` (page 67) representing the record that has an attribute whose value is to be set.

`inAttributeType`
On input, a value of type `tDataNodePtr` that points to a `tDataBuffer` (page 119) structure containing the type of the attribute that is to be set.

`inAttributeValuePtr`
On input, a value of type `tAttributeValueEntryPtr` that points to a `tAttributeEntry` (page 122) containing the attribute value that is to be set.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsSetAttributeValue` function sets the specified attribute to the specified value.

Directory Services Utility Functions

This section describes Directory Services utility functions. The functions are organized as follows:

- "Attribute Utility Functions" (page 85), which describes utility functions for attributes.
- "Data Buffer Utility Functions" (page 87), which describes utility functions for working with data buffers.
- "Data List Utility Functions" (page 89), which describes utility functions for working with data lists.
- "Data Node Utility Functions" (page 102), which describes utility functions for working with data nodes.
- "Miscellaneous Utility Functions" (page 106), which describes utility functions for sending data directly to a plug-in and for deallocating continuation data.

Attribute Utility Functions

This section describes utility functions that support creating attributes and setting the value of attributes. The functions are

- `dsAllocAttributeValueEntry` (page 86), which allocates a `tAttributeValueEntry` structure and returns a pointer to it.

- `dsDeallocAttributeValueEntry` (page 87), which deallocates `tAttributeValueEntry` structures allocated by `dsAllocAttributeValueEntry`.

dsAllocAttributeValueEntry

Allocates an attribute value entry structure having the specified value.

```
tAttributeValueEntryPtr dsAllocAttributeValueEntry(
                    tDirReference inDirReference,
                    unsigned long inAttrValueID
                    void *inAttrValueData,
                    unsigned long inAttrValueDataLen);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inAttrValueID` On input, a value of type `unsigned long` containing an attribute value ID.

`inAttrValueData` On input, a pointer an arbitrary value containing an attribute value.

`inAttrValueDataLen`
On input, the length of valid data in the value pointed to by `inAttrValueData`.

function result A value of type `tAttributeValueEntryPtr` that points to the new `tAttributeValueEntry` (page 122) structure.

DISCUSSION

The `dsAllocAttributeValueEntry` function allocates a structure of type `tAttributeValueEntry` (page 122) and returns a pointer to it. The allocated structure contains the attribute value ID and attribute value specified by the `inAttrValueID`, `inAttrValueData`, and `inAttrValueDataLen` parameters.

To set the value of an attribute, call `dsSetAttributeValue` (page 84) and pass the attribute value entry pointer returned by `dsAllocAttributeValueEntry`.

To release the memory associated with `tAttributeValueEntryPtr`, call `dsDeallocAttributeValueEntry` (page 87).

dsDeallocAttributeValueEntry

Deallocates an attribute value entry.

```
tDirStatus dsDeallocAttributeValueEntry(
                    tDirReference inDirRef,
                    tAttributeValueEntryPtr inAttrValueEntry);
```

`inDirRef` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inAttrValueEntry` On input, a value of type `tAttributeValueEntryPtr` that points to the `tAttributeValueEntry` (page 122) structure that is to be deallocated.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDeallocAttributeValueEntry` function deallocates an attribute value entry structure and the pointer to it that were previously allocated by calling `dsAllocAttributeValueEntry` (page 86).

Data Buffer Utility Functions

This section describes utility functions that support creating and managing data buffers. The functions are

- `dsDataBufferAllocate` (page 88), which allocates a directory services data buffer.
- `dsDataBufferDeAllocate` (page 88), which deallocates a directory services data buffer.

dsDataBufferAllocate

Allocates a Directory Services data buffer.

```
tDataBufferPtr dsDataBufferAllocate(
                    tDirReference inDirReference,
                    unsigned long inBufferSize);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inBufferSize` On input, a value of type `unsigned long` that specifies the length of the buffer that is to be allocated.

function result A value of type `tDataBufferPtr` that points to the allocated `tDataBuffer` (page 119) structure.

DISCUSSION

The `dsDataBufferAllocate` function allocates a Directory Services data buffer of the specified size and returns a value of type `tDataBufferPtr` that points to the allocated buffer.

Directory Services data buffers are used by many Directory Services functions to exchange information between the calling program and Directory Services.

When you no longer need a Directory Services data buffer, call `dsDataBufferDeAllocate`(page 88) to deallocate the memory that is associated with it.

dsDataBufferDeAllocate

Deallocates a Directory Services data buffer.

```
tDirStatus dsDataBufferDeAllocate(
                    tDirReference inDirReference,
                    tDataBufferPtr inDataBufferPtr);
```

`inDirReference`
: On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46) for which a data buffer is to be deallocated

`inDataBufferPtr`
: A value of type `tDataBufferPtr` that points to the `tDataBuffer` (page 119) structure that is to be deallocated.

function result
: A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataBufferDeAllocate` function deallocates a Directory Services data buffer that was previously allocated by `dsDataBufferAllocate` (page 88).

Data List Utility Functions

This section describes utility functions that support creating and managing data lists. The functions are

- `dsAppendStringToList` (page 90), which appends a string to a data list.
- `dsBuildListFromNodes` (page 91), which builds a data list from the specified nodes.
- `dsBuildListFromNodesAlloc` (page 91), which builds a data list using a data list that has already been allocated.
- `dsBuildFromPath` (page 92), which builds a data list from a path.
- `dsBuildListFromStrings` (page 93), which builds a data list from a specified string.
- `dsBuildListFromStringsAlloc` (page 94), which builds a data list from a specified string using a data list that has already been allocated.
- `dsDataListCopyList` (page 95), which copies a list of data nodes.
- `dsDataListDeAllocate` (page 96), which deallocates a data list.
- `dsDataListGetNodeAlloc` (page 97), which obtains a node from a list of data nodes.

- dsDataListGetNodeCount (page 98), which obtains the number of nodes in a list of data nodes.
- dsDataListInsertNode (page 98), which inserts a node into a list of data nodes.
- dsDataListMergeList (page 99), which merges two lists of data nodes.
- dsDataListRemoveNodes (page 100), which removes a node from a list of data nodes by node name.
- dsDataListThisNode (page 100), which removes a node from a list of data nodes by index value.
- dsGetDataLength (page 101) , which the length of data in a data list.
- dsGetPathFromList (page 102), which gets the path from a data list.

dsAppendStringToList

Appends the specified string to a data list.

```
tDirStatus dsAppendStringToList(
                    tDataListPtr inDataList,
                    char *inCString);
```

inDataList	On input, a value of type tDataListPtr that points to the data list to which the string specified by inCString is to be appended.
inCString	On input, a pointer to a null-terminated string containing the value that is to be appended to the data list.
function result	A value of type tDirStatus indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The dsAppendStringToList function appends the specified string to the specified data list.

dsBuildListFromNodes

Builds a data list from one or more data nodes.

```
tDataListPtr dsBuildListFromNodes(
                    tDirReference inDirReference,
                    tDataNodePtr in1stDataNodePtr,
                    ...);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`in1stDataNodePtr`
On input, a value of type `tDataNodePtr` that points to a data node that is to be incorporated into the data list. The `in1stDataNodePtr` parameter may be followed by one or more parameters of type `tDataNodePtr`, each pointing to a data node. The data node may have been allocated by calling `dsDataNodeAllocateBlock` (page 103) or `dsDataNodeAllocateBlock` (page 103).

function result A value of type `tDirListPtr` that points to the new data list.

DISCUSSION

The `dsBuildListFromNodes` function uses the specified data nodes to build a null-terminated data list.

When you no longer need the data list, call `dsDataListDeAllocate` (page 96) to release the memory that is associated with it.

dsBuildListFromNodesAlloc

Fills in a previously allocated data list using data nodes.

```
tDirStatus dsBuildListFromNodesAlloc(
                    tDataListPtr inDataList,
                    tDataNodePtr in1stDataNodePtr,
                    ...);
```

`inDataList` On input, a value of type `tDataListPtr` that points to a previously allocated data list. The data list may have been allocated by, for example, `dsDataNodeAllocateBlock` (page 103).

`in1stDataNodePtr`
On input, a value of type `tDataNodePtr` that points to a data node. The `in1stDataNodePtr` parameter may be followed by one or more parameters of type `tDataNodePtr`, each pointing to a data node. The data node may have been allocated by calling `dsDataNodeAllocateBlock` (page 103) or `dsDataNodeAllocateBlock` (page 103).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsBuildListFromNodesAlloc` function fills in a previously allocated data list with information from the specified data nodes. The resulting data list is null-terminated.

When you no longer need the data list, call `dsDataListDeAllocate` (page 96) to release the memory associated with it.

dsBuildFromPath

Builds a data list from a path.

```
tDataListPtr dsBuildFromPath(
                    tDirReference inDirReference,
                    char *inPathCString,
                    char *inPathSeparatorCString);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inPathCString` On input, a pointer to a null-terminated string containing a path.

`inPathSeparatorCString`
On input, a pointer to a null-terminated string containing the character that delimits the components of the path specified by `inPathCString`.

function result A value of type `tDirListPtr` that points to the new data list.

DISCUSSION

The `dsBuildFromPath` function uses a path to build a null-terminated data list and returns a pointer to it. Many Directory Services functions take a pointer to a data list as a parameter. For example, you can pass the resulting data list pointer as a parameter to `dsOpenDirNode` (page 58).

When you no longer need the data list, call `dsDataListDeAllocate` (page 96) to release the memory associated with it.

dsBuildListFromStrings

Builds a data list from strings.

```
tDataListPtr dsBuildListFromStrings(
                    tDirReference inDirReference,
                    char *in1stCString,
                    ...);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`in1stCString` On input, a pointer to a null-terminated string containing data that is to be added to the data list. The `in1stCString` parameter may be followed by one or more parameters of type `char *`, each pointing to a C string containing data that is to be added to the data list.

function result A value of type `tDataListPtr` that points to the `tDataList` (page 120) structure that has been created.

DISCUSSION

The `dsBuildListFromStrings` function uses the contents of one or more null-terminated strings to build a data list and returns a pointer to it.

When you no longer need the data list, call `dsDataListDeAllocate` (page 96) to release the memory associated with it.

dsBuildListFromStringsAlloc

Fills in a previously allocated data list using data from strings.

```
tDirStatus dsBuildListFromStringsAlloc(
                    tDataListPtr inDataList,
                    const char *in1stCString,
                    ...);
```

`inDataList` On input, a value of type `tDataListPtr` that points to a previously allocated data list. The data list, for example, may have been created by `dsDataNodeAllocateBlock` (page 103).

`in1stCString` On input, a pointer to a character string that specifies the name of a data node to add to the data list. The `in1stCString` parameter may be followed by one or more additional parameters of type `char *`, each pointing to a C string containing data that is to be added to the data list.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsBuildListFromStringsAlloc` function fills in a data list using the data contained by the specified null-terminated strings.

dsDataListAllocate

Allocates a data list.

```
tDataListPtr dsDataListAllocate(tDirReference inDirReference);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

function result A value of type `tDataListPtr` that points to the allocated `tDataList` (page 120) structure. If `dsDataListAllocate` cannot allocate the data list, it returns `NULL`.

DISCUSSION

The `dsDataListAllocate` function allocates an empty data list and returns a value of type `tDataListPtr` that points to it.

Many Directory Services functions return information in a data list, such as `dsGetDirNodeName` (page 57), and receive information in a data list, such as `dsFindDirNodes` (page 52), `dsGetDirNodeInfo` (page 54), `dsGetRecordList` (page 64), and `dsDoAttributeValueSearch` (page 74).

To add add data to the datalist, call `dsBuildListFromNodesAlloc` (page 91) or `dsBuildListFromStringsAlloc` (page 94).

When you no longer need the data list, call `dsDataListDeAllocate` (page 96) to release the memory associated with it.

dsDataListCopyList

Copies a data list.

```
tDataListPtr dsDataListCopyList(
                    tDirReference inDirReference,
                    tDataListPtr inDataListSource);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inDataListSource`
On input, a value of type `tDataListPtr` pointing to the data list to be copied.

function result A value of type `tDataListPtr` that points to the copy of the data list. If `dsDataListCopyList` cannot copy the list, it returns `NULL`.

DISCUSSION

The `dsDataListCopyList` function copies a data list and returns a pointer to the copy of the data list.

dsDataListDeAllocate

Deallocates a data list.

```
tDirStatus dsDataListDeAllocate(
                    tDirReference inDirReference,
                    tDataListPtr inDataList,
                    Boolean inDeAllocateNodesFlag);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inDataList` On input, a value of type `tDataListPtr` pointing to the `tDataList` (page 120) structure that is to be deallocated.

`inDeAllocateNodesFlag`
On input, a Boolean value that specifies deallocation flags. Set `inDeAllocateNodesFlag` to `TRUE` if you want to deallocate the list and the nodes in the list. Set `inDeAllocateNodesFlag` to `FALSE` if you want to deallocate the list without deallocating the nodes in the list.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataListDeAllocate` function deallocates a data list previously created by calling `dsDataNodeAllocateBlock` (page 103), `dsBuildListFromNodes` (page 91), `dsBuildFromPath` (page 92), `dsBuildListFromStrings` (page 93), or `dsDataListCopyList` (page 95).

You should call `dsDataListDeAllocate` to release the memory associated with a data list when you have no further need for a data list.

dsDataListGetNodeAlloc

Obtains a data node from a data list.

```
tDirStatus dsDataListGetNodeAlloc(
                    tDirReference inDirReference,
                    tDataListPtr inDataListPtr,
                    unsigned long inNodeIndex,
                    tDataNodePtr* outDataNode);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inDataList` On input, a value of type `tDataListPtr` that points to the data list from which a data node is to be obtained.

`inNodeIndex` On input, a value of type `unsigned long` that identifies the data node to obtain. Set `inNodeIndex` to 1 to get the first node. Set `inNodeIndex` to 2 to get the second node, and so on.

`outDataNode` On output, a value of type `tDataNodePtr` (page 119) that points to the data node obtained from the data list.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataListGetNode` function obtains a data node from a data list.

Note
The `dsDataListGetNodeAlloc` function replaces the `dsDataListGetNode` function which is obsolete. ◆

dsDataListGetNodeCount

Obtains the number of nodes in a data list.

```
unsigned long dsDataListGetNodeCount(tDataListPtr inDataList);
```

`inDataListPtr` On input, a value of type `tDataListPtr` pointing to a data list containing the names of nodes that are to be counted.

function result The number of nodes in the data list or an error code. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataListGetNodeCount` function obtains the number of nodes in a data list.

dsDataListInsertNode

Inserts a node in a data list.

```
tDirStatus dsDataListInsertNode(
                    tDataListPtr inDataList,
                    tDataNodePtr inAfterDataNode,
                    tDataNodePtr inInsertDataNode);
```

`inDataList` On input, a value of type `tDataListPtr` pointing to a data list containing a list of nodes.

`inAfterDataNode` On input, a value of type `tDataNodePtr` pointing to a data node containing the node name in `inDataList` after which the node pointed to by `inInsertDataNode` is to be inserted.

`inInsertDataNode` On input, a value of type `tDataNodePtr` pointing to a data node containing the name of the node that is to be inserted.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see “Result Codes” (page 130).

DISCUSSION

The `dsDataListInsertNode` function inserts a node into a list of nodes.

dsDataListMergeList

Merges two data lists.

```
tDirStatus dsDataListMergeList(
                    tDataListPtr inDataList,
                    tDataNodePtr inAfterDataNode,
                    tDataListPtr inMergeDataList);
```

`inDataList` On input, a value of type `tDataListPtr` pointing to a data list containing data nodes.

`inAfterDataNode` On input, a value of type `tDataNodePtr` that points to a data node pointing to the name of the node in `inDataList` after which the nodes pointed to by `inMergeDataList` are to be merged.

`inMergeDataList` On input, a value of type `tDataListPtr` pointing to the data list that is to be merged with the list pointed to by `inDataList`.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see “Result Codes” (page 130).

DISCUSSION

The `dsDataListMergeList` function merges two data lists. The nodes in the data list pointed by the `inMergeDataList` parameter are merged with the nodes in the datalist pointed to by the `inDataList` parameter after the node pointed to by the `inAfterDataNode` parameter.

dsDataListRemoveNodes

Removes data nodes from a data list.

```
tDirStatus dsDataListRemoveNodes(
                    tDataListPtr inDataList,
                    tDataNodePtr in1stDataNode,
                    unsigned long inDeleteCount);
```

`inDataList` On input, a value of type `tDataListPtr` pointing the data list from which nodes are to be removed.

`in1stDataNode` On input, a value of type `tDataNodePtr` pointing to the data node that specifies the first data node that is to be removed.

`inDeleteCount` On input, a value of type `unsigned long` that specifies the number of data nodes to remove.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataListRemoveNodes` function removes data nodes from a data list. The `in1stDataNode` parameter points to a data node that contains the name of the first data node to be removed. The `inDeleteCount` parameter specifies the total number of data nodes that are to be removed including the first data node.

dsDataListThisNode

Removes data nodes from a data list.

```
tDirStatus dsDataListThisNode(
                    tDataListPtr inDataList,
                    unsigned long inNodeIndex,
                    unsigned long inDeleteCount);
```

`inDataList` On input, a value of type `tDataListPtr` pointing to the data list from which data nodes are to be removed.

`inNodeIndex`	On input, a value of type `unsigned long` that identifies the data node to remove. Set `inNodeIndex` to 1 to remove the first node. Set `inNodeIndex` to 2 to remove the second node, and so on.
`inDeleteCount`	On input, a value of type `unsigned long` that specifies the total number of data nodes to remove.
function result	A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataListThisNode` function removes data nodes from a data list. The `inNodeIndex` parameter specifies the index of the first data node that is to be removed. The `inDeleteCount` parameter specifies the total number of data nodes that are to be removed.

dsGetDataLength

Obtains the length of data in a data list.

```
unsigned long dsDataList dsGetDataLength(tDataListPtr inDataList);
```

`inDataListPtr`	On input, a value of type `tDataListPtr` pointing to the `tDataList` (page 120) structure whose length is to be obtained.
function result	The length of data in the specified data list or an error code. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetDataLength` function obtains the length in bytes of data in a data list.

dsGetPathFromList

Gets the path from a data list.

```
char* dsGetPathFromList(
                    tDirReference inDirReference,
                    tDataListPtr inDataList,
                    char *inDelimiter);
```

`inDirReference` On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inDataList` On input, a value of type `tDataListPtr` pointing to a data list.

`inDelimiter` On input, a pointer to a character string containing the character that delimits the components of the path in the `inDataList` parameter.

function result A pointer to a character string that contains the path that was obtained from the data list.

DISCUSSION

The `dsGetPathFromList` function gets the path from a data list.

Data Node Utility Functions

This section describes utility functions that support creating and managing data nodes. The functions are

- `dsDataNodeAllocateBlock` (page 103), which allocates a data node.
- `dsDataNodeAllocateString` (page 104), which allocates a data node.
- `dsDataNodeDeAllocate` (page 105), which deallocates a data node.
- `dsDataNodeGetSize` (page 105), which gets the size of a data node.
- `dsDataNodeSetLength` (page 106), which sets the length of a data node.

dsDataNodeAllocateBlock

Allocates a data node using a `tbuffer` data type.

```
tDataNodePtr dsDataNodeAllocateBlock(
                    tDirReference inDirReference,
                    unsigned long inDataNodeSize,
                    unsigned long inDataNodeLength,
                    tBuffer inDataNodeBuffer);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46) that represents the directory system in which the node is to be allocated.

`inDataNodeSize`
On input, a value of type `unsigned long` that specifies the size of `inDataNodeBuffer`.

`inDataNodeLength`
On input, a value of type `unsigned long` that specifies the length of valid data in `inDataNodeBuffer`.

`inDataNodeBuffer`
On input, a value of type `tBuffer` containing the value the data node is to contain.

function result A value of type `tDataNodePtr` that points to the allocated data node and that can be passed as a parameter to Directory Services functions that require a value of type `tDataNodePtr` as a parameter. If `dsDataNodeAllocateBlock` cannot allocate the data node, it returns `NULL`.

DISCUSSION

The `dsDataNodeAllocateBlock` function uses a `tbuffer` data type to allocate a Directory Services data node and returns a pointer to the allocated data node. Use the data node as a convenient way to pass data, such as record names and authentication types, to Directory Services functions.

To release the memory associated with a data node, call `dsDataNodeDeAllocate` (page 105).

To use a C string to allocate a data node, call `dsDataNodeAllocateString` (page 104).

dsDataNodeAllocateString

Allocates a data node using a C string.

```
tDataNodePtr dsDataNodeAllocateString(
                    tDirReference inDirReference,
                    char *inCString);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inCString` A pointer to a value of type `char` that specifies the value the data node is to contain.

function result A value of type `tDataNodePtr` that points to the allocated data node and that can be passed as a parameter to Directory Services functions that require a value of type `tDataNodePtr` as a parameter. If `dsDataNodeAllocateString` cannot allocate the data node, it returns `NULL`.

DISCUSSION

The `dsDataNodeAllocateString` function uses a C string to allocate a Directory Services data node and returns a pointer to the allocated data node. Use the data node as a convenient way to pass data, such as record names and authentication types, to Directory Services functions.

To release the memory assocated with a data node, call `dsDataNodeDeAllocate` (page 105).

To use a `tbuffer` to allocate a data node, call `dsDataNodeAllocateBlock` (page 103).

dsDataNodeDeAllocate

Deallocates a data node.

```
tDirStatus dsDataNodeDeAllocate(
                    tDirReference inDirReference,
                    tDataNodePtr inDataNodePtr);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inDataNodePtr` On input, a value of type `tDataNodePtr` that points to the `tDataBuffer` (page 119) structure that is to be deallocated.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataNodeDeAllocate` function deallocates a Directory Services data node that was created by previously calling `dsDataNodeAllocateBlock` (page 103) or `dsDataNodeAllocateString` (page 104).

dsDataNodeGetSize

Obtains the size of a data node's buffer.

```
unsigned long dsDataNodeGetSize(tDataNodePtr inDataNodePtr);
```

`inDataNodePtr` On input, a value of type `tDataNodePtr` that points to the `tDataBuffer` (page 119) structure whose buffer size is to be obtained.

function result A value of type `unsigned long` that contains the length of the buffer. If `dsDataNodeGetSize` cannot obtain the length of the buffer, it returns 0.

DISCUSSION

The `dsDataNodeGetSize` function obtains the size of a data node's buffer.

dsDataNodeSetLength

Sets the length valid data in a data node's buffer.

```
tDirStatus dsDataNodeSetLength(
                    tDataNodePtr inDataNodePtr,
                    unsigned long inDataNodeLength);
```

`inDataNodePtr` On input, a value of type `tDataNodePtr` that points to the data node whose buffer size is to be set.

`inDataNodeLength`
On input, a value of type `unsigned long` that specifies the length of valid data in the buffer.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDataNodeSetLength` function sets the length of valid data in the buffer of the data node pointed to by `inDataNodePtr`.

Miscellaneous Utility Functions

This section describes miscellaneous utility functions. The functions are

- `dsDoPluginCustomCall` (page 107), which sends information directly to a Directory Services plug-in.
- `dsReleaseContinueData` (page 108), which releases memory allocated for continuation data.

dsDoPluginCustomCall

Sends information directly to a Directory Services plug-in.

```
tDirStatus dsDoPlugInCustomCall(
                    tDirNodeReference inDirNodeReference,
                    unsigned long inCustomRequestCode,
                    tDataBufferPtr inCustomRequestData,
                    tDataBufferPtr outCustomRequestResponse);
```

`inDirNodeReference`
On input, a value of type `tDirNodeReference`, obtained by calling `dsOpenDirNode` (page 58), that identifies the plug-in to which information is to be sent.

`inCustomRequestCode`
On input, a value of type `unsigned long`, containing a request code that is to be sent to the plug-in.

`inCustomRequestData`
On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure containing data that is to be sent to the plug-in.

`outCustomRequestResponse`
On input, a value of type `tDataBufferPtr` created by calling `dsDataBufferAllocate` (page 88) pointing to a `tDataBuffer` (page 119) structure. On output, `outCustomRequestResponse` contains the plug-in's response to the information that was sent.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsDoPlugInCustomCall` function sends information directly to the Directory Services plug-in for the node represented by `inDirNodeReference`.

dsReleaseContinueData

Releases memory allocated for continuation data.

```
tDirStatus dsReleaseContinueData(
                    tDirReference inDirReference,
                    tContextData inContinueData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inContinueData`
On input, a value of type `tContextData` that is to be released. On output, `inContinueData` is `NULL`.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsReleaseContinueData` function releases memory allocated for continuation data, which may be returned by the `dsDoDirNodeAuth` (page 50), `dsFindDirNodes` (page 52), `dsGetDirNodeInfo` (page 54), `dsGetDirNodeList` (page 56), `dsGetRecordList` (page 64), `dsDoAttributeValueSearch` (page 74), `dsGetAttributeValue` (page 76), and `dsDoAttributeValueSearch` (page 74).

Managing Custom Routines

This section describes Directory Services functions for managing custom memory and thread routines.

- `dsGetCustomAllocate` (page 109) gets the address of a custom memory allocation routine.
- `dsRegisterCustomMemory` (page 110) registers a custom memory allocation routine.
- `dsUnRegisterCustomMemory` (page 111) unregisters a custom memory allocation routine.
- `dsGetCustomThread` (page 112) gets the address of a custom thread routine.

- `dsRegisterCustomThread` (page 113) registers a custom thread routine.
- `dsUnRegisterCustomThread` (page 114) unregisters a custom thread routine.

Note
Custom memory and custom thread routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsGetCustomAllocate

Get custom memory routines.

```
tDirStatus dsGetCustomAllocate(
                    tDirReference inDirReference,
                    fpCustomAllocate *outCustomAllocate,
                    fpCustomDeAllocate *outCustomDeAllocate,
                    tClientData *outClientData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`outCustomAllocate`
On input, a pointer to a value of type `fpCustomAllocate`. On output, `outCustomAllocate` points to the custom memory allocation routine associated with the specified directory reference.

`outCustomDeAllocate`
On input, a pointer to a value of type `fpCustomDeAllocate`. On output, `outCustomDeAllocate` points to the custom memory deallocation routine associated with the specified directory reference.

`outClientData` On input, a pointer to a value of type `tClientData`. On output, `tClientData` points to the client data that was previously registered by calling `dsRegisterCustomMemory`.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetCustomMemory` function gets information about the custom memory routines that have been registered for the specified directory reference. For information about custom memory routines, see "Custom Memory Routines" (page 116).

Note
Custom memory routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsRegisterCustomMemory

Register custom memory routines.

```
tDirStatus dsRegisterCustomMemory(
                    tDirReference inDirReference,
                    fpCustomAllocate inCustomAllocate,
                    fpCustomDeAllocate inCustomDeAllocate,
                    tClientData inClientData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inCustomAllocate`
On input, a value of type `fpCustomAllocate` that identifies the custom memory allocation routine that is to be registered.

`inCustomDeAllocate`
On input, a value of type `fpCustomAllocate` that identifies the custom memory allocation routine that is to be registered.

`inClientData` On input, a value of type `tClientData` that is passed to your application's custom memory routine when it is called.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsRegisterCustomMemory` function registers custom memory allocation and deallocation routines. These routines should be used along with API dispatch related to the specified `tDirReference`. For information about custom memory routines, see "Custom Memory Routines" (page 116).

Note
Custom memory routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsUnRegisterCustomMemory

Unregister custom memory allocation and deallocation routines.

```
tDirStatus dsUnRegisterCustomMemory(
                    tDirReference inDirReference,
                    fpCustomAllocate inCustomAllocate,
                    fpCustomDeAllocate inCustomDeAllocate,
                    tClientData inClientData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inCustomAllocate`
On input, a value of type `fpCustomAllocate` that identifies the custom memory allocation routine that is to be unregistered.

`inCustomDeAllocate`
On input, a value of type `fpCustomAllocate` that identifies the custom memory deallocation routine that is to be unregistered.

`inClientData` On input, a value of type `tClientData` that is passed to the custom memory routines before they are unregistered.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsUnRegisterCustomMemory` function unregisters the specified custom memory allocation and deallocation routines for the specified directory reference. For information about custom memory routines, see "Custom Memory Routines" (page 116).

Note
Custom memory routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsGetCustomThread

Gets information about custom thread routines.

```
tDirStatus dsGetCustomThread(
                    tDirReference inDirReference,
                    fpCustomThreadBlock *outCustomBlock,
                    fpCustomThreadUnBlock *outCustomUnBlock,
                    fpCustomThreadYield *outCustomYield,
                    tClientData *outClientData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`outCustomBlock`
On input, a pointer to a value of type `fpCustomThreadBlock`. On output, `outCustomAllocate` points to the custom block routine associated with the specified directory reference.

`outCustomUnBlock`
On input, a pointer to a value of type `fpCustomThreadUnBlock`. On output, `outCustomDeAllocate` points to the custom unblock routine associated with the specified directory reference.

`outCustomYield`
On input, a pointer to a value of type `fpCustomThreadYield`. On output, `outCustomYield` points to the custom yield routine associated with the specified directory reference.

`outClientData` On input, a pointer to a value of type `tClientData`. On output, `tClientData` points to the client data that was previously registered by calling `dsRegisterCustomThread`(page 113).

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsGetCustomThread` function gets information about the custom thread routines that have been registered for the specified directory reference. For information about custom thread routines, see "Custom Thread Routines" (page 116).

Note
Custom thread routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsRegisterCustomThread

Registers custom thread routines.

```
tDirStatus dsRegisterCustomThread(
                    tDirReference inDirReference,
                    fpCustomThreadBlock inCustomBlock,
                    fpCustomThreadUnBlock inCustomUnBlock,
                    fpCustomThreadYield inCustomYield,
                    tClientData inClientData);
```

`inDirReference`
On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inCustomBlock`
On input, a value of type `fpCustomThreadBlock` that identifies the custom block routine that is to be registered.

`inCustomUnBlock`
On input, a value of type `fpCustomThreadUnBlock` that identifies the custom unblock routine that is to be registered.

`inCustomYield`
: On input, a value of type `fpCustomThreadYield` that identifies the custom yield routine that is to be registered.

`inClientData`
: On input, a value of type `tClientData` that is passed to the custom block, unblock, and yield routines as part of the registration process. The value of `inClientData` is returned by the `dsGetCustomThread` function.

function result
: A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsRegisterCustomThread` function registers custom block, unblock, and yield routines. For information about custom thread routines, see "Custom Thread Routines" (page 116).

Note
Custom thread routines are supported in Mac OS 8 and Mac OS 9 only. ◆

dsUnRegisterCustomThread

Deregisters custom thread routines.

```
tDirStatus dsUnRegisterCustomThread(
                    tDirReference inDirReference,
                    fpCustomThreadBlock inCustomBlock,
                    fpCustomThreadUnBlock inCustomUnBlock,
                    fpCustomThreadYield inCustomYield,
                    tClientData inClientData;
```

`inDirReference`
: On input, a value of type `tDirReference` (page 117) obtained by calling `dsOpenDirService` (page 46).

`inCustomBlock`
: On input, a value of type `fpCustomThreadBlock` that identifies the custom block routine that is to be registered.

`inCustomUnBlock`
On input, a value of type `fpCustomThreadUnBlock` that identifies the custom unblock routine that is to be registered.

`inCustomYield`
On input, a value of type `fpCustomThreadUnBlock` that identifies the custom yield routine that is to be registered.

`inClientData` On input, a value of type `tClientData` that is passed to the custom block, unblock, and yield routines before they are unregistered.

function result A value of type `tDirStatus` indicating success or an error. For a list of possible result codes, see "Result Codes" (page 130).

DISCUSSION

The `dsUnRegisterCustomThread` function unregisters the specified custom thread routines for the specified directory reference. For information about custom thread routines, see "Custom Thread Routines" (page 116).

Note
Custom thread routines are supported in Mac OS 8 and Mac OS 9 only. ◆

Application-Defined Routines

This section describes five dispatch primitives that can be overridden by custom routines provided by a Directory Services client. The five routines are:

- `fpCustomAllocate`, a custom memory allocation routine
- `fpCustomDeAllocate`, a custom memory deallocation routine
- `fpCustomThreadBlock`, a custom thread blocking routine
- `fpCustomThreadUnBlock`, a custom thread unblocking routine
- `fpCustomThreadYield`, a custom thread yield routine

For Mac OS 9 applications that have custom run-time requirements, Directory Services provides function pointers for overriding the standard Mac OS memory allocation, deallocation, thread block, thread unblock, and thread yield

functions . Custom run-time requirements include situations when standard Mac OS features are not available at API dispatch time.

The Directory Services API is a Mac OS 9 API, which means that Directory Services is compatible with Mac OS 9 and with Mac OS X. Mac OS X applications will not need to override the standard Mac OS functions and most Mac OS 9 applications will not need to override them, either.

Custom Memory Routines

This section describes function pointers for custom memory allocation and deallocation routines. These routines only need to be set if for some reason the standard OS memory allocation and deallocation routines aren't appropriate. If these routines are not set, the standard OS allocation and deallocation routines are used.

```
typedef tDirStatus (*fpCustomAllocate) (
    tDirReference inDirReference,
    tClientData inClientData,
    unsigned long inAllocationRequest,
    tBuffer *outAllocationPtr);

typedef tDirStatus (*fpCustomDeAllocate) (
    inDirReference nDirReference,
    tClientData nClientData,
    tBuffer inAllocationPtr);
```

Custom Thread Routines

This section describes function pointers for custom thread routines for blocking, unblocking, and yielding. These routines only need to be set if for some reason the standard OS functions aren't appropriate. If these routines are not set, the standard OS blocking, unblocking, and yielding functions are used.

```
typedef tDirStatus (*fpCustomThreadBlock) (
    tDirReference inDirReference,
    tClientData inClientData);
```

```
typedef tDirStatus (*fpCustomThreadUnBlock) (
    tDirReference inDirReference,
    tClientData inClientData);

typedef tDirStatus (*fpCustomThreadYield) (
    tDirReference inDirReference,
    tClientData inClientData);
```

Directory Services Structures and Other Data Types

The following structures supply information needed to call Directory Services functions:

- The data type `tDirReference` (page 117) is a reference to an open session with Directory Services.
- The structure `tDataBuffer` (page 119) is used by functions that operate on data nodes.
- The structure `tDataList` (page 120) is used to store an ordered list of `tDataNode` structures.
- The structure `tAccessControlEntry` (page 121) is used to store access controls for data nodes.
- The structure `tAttributeValueEntry` (page 122) is used to return the response for any data fetch request.
- The structure `tAttributeEntry` (page 122) stores information about an attribute.
- The structure `tRecordEntry` (page 123) is used to return lists of directory node records.

tDirReference

The `tDirReference` data type is an `unsigned long` that refers to an open Directory Services session. Most Directory Services functions require a

tDirReference as a parameter. The tDirReference data type is defined as follows:

```
typedef unsigned long tDirReference;
```

You obtain a tDirReference by calling dsOpenDirService (page 46) to open a Directory Services session. You call dsCloseDirService (page 46) to close the session and dispose of the reference when you no longer need it.

tDirNodeReference

The tDirNodeReference data type is an unsigned long that refers to an open session with directory node. Directory Services functions that operate on nodes, records, and attributes require a tDirNodeReference as a parameter. The tDirNodeReference data type is defined as follows:

```
typedef unsigned long tDirNodeReference;
```

You obtain a tDirNodeReference by calling dsOpenDirNode (page 58) to open a session with a directory node. You call dsCloseDirNode (page 47) to close the session and dispose of the reference when you no longer need it.

tClientData

The tClientData data type is pointer to an arbitrary value used to pass data to custom thread and custom memory routines. The tClientData data type is defined as follows:

```
typedef void * tClientData;
```

tBuffer

The `tBuffer` data type is pointer to an arbitrary value used pass data to custom thread and custom memory routines. The `tClientData` data type is defined as follows:

```
typedef void * tBuffer;
```

tContextData

The `tContextData` data type is pointer to an arbitrary value used exchange continuation data between an application and Directory Services. For example, when the results of calling a Directory Services function exceed the size of the response buffer, the function returns a value of type `tContextData`. Your application can get the next buffer of results by calling the function again and passing the context data as a parameter.

The `tContextData` data type is defined as follows:

```
typedef void * tContextData;
```

tDataBuffer

The `tDataBuffer` structure provides a standard format for passing information to Directory Services functions. It typically contains strings, directory path nodes, and attribute types.

```
typedef struct
{
    unsigned long fBufferSize;
    unsigned long fBufferLength;
    char fBufferData[1];
} tDataBuffer;
typedef tDataBuffer *tDataBufferPtr;
typedef tDataBuffer tDataNode;
typedef tDataNode *tDataNodePtr;
```

Field descriptions

`fBufferSize`	The number of bytes allocated for this structure. The value of `fBufferSize` should be set when `tDataBuffer` is created.
`fBufferLength`	The number of meaningful bytes in `fBufferData`. You should call `dsDataNodeSetLength` (page 106) to adjust this value each time you change the value of the `fBufferData` field.
`fBufferData`	An array of characters.

Call `dsDataNodeAllocateBlock` (page 103) or `dsDataNodeAllocateString` (page 104) to allocate a data node. Call `dsDataNodeDeAllocate` (page 105) to release the memory associated with a data node when it is no longer needed.

tDataNode

The `tDataNode` data type is a `tDataBuffer` (page 119) that provides a standard format for passing information to Directory Services functions. It is typically used to contain strings, directory path nodes, and attribute types that are exchanged between a program and Directory Services.

```
typedef tDataBuffer tDataNode;
typedef tDataNode *tDataNodePtr;
```

Call `dsDataNodeAllocateBlock` (page 103) or `dsDataNodeAllocateString` (page 104) to allocate a data node.

Call `dsDataNodeDeAllocate` (page 105) to release the memory associated with a data node when it is no longer needed.

tDataList

The `tDataList` structure is an ordered collection of `tDataNode` structures.

```
typedef struct
{
    unsigned long fDataNodeCount;
```

```
    tDataNodePtr fDataListHead;
} tDataList;
typedef tDataList *tDataListPtr;
```

Field descriptions

`fDataNodeCount` The number of data nodes in this data list structure.

`fDataListHead` The first pointer to a data node in this data list structure.

The `tDataList` structure is used to represent lists of items, such as directory nodes, full path names, attribute type lists, and lists of record names.

To allocate a data list, call `dsDataListAllocate` (page 95). To build a data list from one ore more data nodes, call `dsBuildListFromNodes` (page 91); to build a data list from one or more C strings, call `dsBuildListFromStrings` (page 93). Or copy a data list by calling `dsDataListCopyList` (page 95). For a complete list of data list manipulation functions, see "Data List Utility Functions" (page 89).

To release the memory associated with a data list when it is no longer needed, call `dsDataListDeAllocate` (page 96).

tAccessControlEntry

The `tAccessControlEntry` structure is the defined format for access controls.

```
typedef struct
{
    unsigned long fGuestAccessFlags;
    unsigned long fDirMemberFlags;
    unsigned long fDirNodeMemberFlags;
    unsigned long fOwnerFlags;
    unsigned long fAdministratorFlags;
} tAccessControlEntry;
typedef tAccessControlEntry *tAccessControlEntryPtr;
```

Each of the bits in the fields represents certain capabilities for the given class of access request. The actual field values will be defined in a future version of Directory Services.

tAttributeValueEntry

The `tAttributeValueEntry` structure is used to get and set the value of an attribute by attribute value ID for attributes that can have more than one value.

```
typedef struct
{
    unsigned long fAttributeValueID;
    tDataNode fAttributeValueData;
} tAttributeValueEntry;
typedef tAttributeValueEntry *tAttributeValueEntryPtr;
typedef unsigned long tAttributeValueListRef;
```

Field descriptions

`fAttributeValueID` A unique ID for this attribute value.

`fAttributeValueData` A `tDataNode` structure containing the value of this attribute.

tAttributeEntry

The `tAttributeEntry` structure stores information about an attribute.

```
typedef struct
{
    unsigned long fAttributeFlags;
    tAccessControlEntry fAttributeAccess;
    unsigned long fAttributeValueCount;
    unsigned long fAttributeDataSize;
    unsigned long fAttributeValueMaxSize;
    tDataNode fAttributeSignature;
} tAttributeEntry;
typedef tAttributeEntry *tAttributeEntryPtr;
typedef unsigned long tAttributeListRef;
```

Field descriptions

`fAttributeFlags` Flags describing this attribute.

`fAttributeAccess` Access controls for this attribute.

`fAttributeValueCount` The number of values associated with this attribute.

`fAttributeDataSize` The total byte count of all attribute values.

`fAttributeValueMaxSize`
The maximum size of a value of this attribute type.

`fAttributeSignature` A byte sequence that uniquely represents this attribute type. The byte sequence is typically a collection of Unicode characters.

Directory Services clients will be able to "discover" the contents of records and directory nodes. One of the items that needs to be discovered is what types of data are contained in a given directory object.

When a Directory Services client application requests the `kDSNAttrSchema` attribute type for a directory node or a record, the response consists of a list of `tAttributeEntry` structures. The list is a table of contents of the directory node or record, telling the client everything it needs to know about the item without having to request the actual data.

tRecordEntry

The `tRecordEntry` structure is used to return the list of records in a directory node.

```
typedef struct
{
    unsigned long fRecordFlags;
    tAccessControlEntry fRecordAccess;
    unsigned long fRecordAttributeCount;
    tDataNode fRecordNameAndType;
}   tRecordEntry;
typedef tRecordEntry *tRecordEntryPtr;
typedef unsigned long tRecordReference;
```

Field descriptions

`fRecordFlags` Record flags.

`fRecordAccess` Access controls for this record.

`fRecordAttributeCount` The number of attribute types.

`fRecordNameAndType` A `tDataNode` containing the record's primary name and its type.

This structure is equivalent to an `FSSpec` in the file system. It contains the high-level basic information about a record.

Directory Services Constants

This section describes constants that have been defined for Directory Services. They are organized into the following sections:

- "Well Known Attribute Type Constants" (page 124)
- "Pattern-Matching Constants" (page 127)

Well Known Attribute Type Constants

This section describes the constants that have been defined for attribute types in this version of Directory Services. Unless otherwise noted, the attributes described in this section can have a single value.

Note
Access controls may prevent any particular client from readin or writing any particular attribute. Some attributes may not be stored and may represent data generated by the Directory Services plug-in on a real-time basis. ◆

`kDSRecordsAll` All records.

`kDSAttributesAll` All attributes.

`kDS1RecordName` Record name attribute.

`kDSSetPasswdBestOf` Directory native authentication.

`kDSStdRecordTypePrefix` The prefix for standard record types.

`kDSNativeRecordTypePrefix`
The prefix for native record types.

`kDSStdRecordTypeUsers` The type for standard user records.

`kDSStdRecordTypeGroups`
The type for standard group records.

`kDSStdRecordTypeMachines`
The type for standard machine records.

`kDSStdRecordTypePrinters`
The type for standard printer records.

`kDSStdRecordTypeMeta`The type for standard meta records.

`kDSStdRecordTypeMeta`The type for standard meta records.

`kDSStdAttrTypePrefix`The prefix for standard attribute types.

`kDSNativeAttrTypePrefix`
The prefix for native attribute types.

`kDSDSAttrNone` The attribute type for no attributes.

`kDSStdAuthMethodPrefix`
The prefix for standard authentication methods.

`kDSNativeAuthMethodPrefix`
The prefix for native authentication methods.

`kDSStdAuthClearText`The standard clear text authentication method.

`kDSStdAuthCrypt` An encryption-based authentication method.

`kDSStdAuthSetPasswd`An authentication method that includes setting the password.

`kDSStdAuthChangePasswd`An authentication method that includes changing the password.

`kDSStdAuthAPOP` The APOP authentication method.

`kDSStdAuth2WayRandom`The Two-Way Random authentication method.

`kDSStdAuthNodeNativeClearTextOK`
A native authentication method that allows clear text authentication.

`kDSStdAuthNodeNativeNoClearText`
A native authentication method that does not allow clear text authentication.

`kDSStdAuthSMB_NT_Key`An SMB authentication method.

`kDSStdAuthSMB_LM_Key`An SMB authentication method.

`kDS1AttrPassword` The password attribute.

`kDS1AttrInternetAlias` The Internet alias attribute.

`kDS1AttrUniqueID` The unique ID attribute.

`kDS1AttrPrimaryGroupID` The primary group ID attribute.

`kDS1AttrMailAttribute` The mail attribute.

`kDS1AttrComment` The comment attribute.

`kDS1AttrRARA` The RARA attribute.

`kDS1AttrGeneratedUID` The generated UID attribute.

`kDS1AttrAdminStatus` The administrator status attribute.

`kDS1AttrPwdAgingPolicy` The password aging policy attribute.

`kDSNAttrRecordAlias` The record alias attribute. Multiple values are allowed.

`kDSNAttrGroupMembership` The group membership attribute. Multiple values are allowed

`kDSNAttrHomeDirectory` The home directory attribute. Multiple values are allowed

`kDSNAttrEMailAddress` The Email address attribute. Multiple values are allowed

`kDSNAttrPhoneNumber` The Phone number attribute. Multiple values are allowed

`kDSNAttrMetaNodeLocation` The meta node location attribute. Multiple values are allowed

`kDS1AttrDataStamp` The data stamp attribute.

`kDS1AttrTotalSize` The total size attribute.

`kDS1AttrTimePackage` The creation, modification, and backup date attribute.

`kDS1AttrAlias` The alias attribute; contains a pointer to another node, record, or attribute.

`kDS1AttrAuthCredential` The authentication credential attribute.

`kDS1AttrCapabilities` The capabilities attribute; clients can use this attribute to find out the capabilities of a directory node.

`kDS1AttrSearchPath` The search path attribute; clients can use this attribute with the search node to get the node's search path.

`kDS1AttrRecordImage` The record image attribute; clients can use this attribute to force the directory service to generate a binary image of the record and all of its attributes.

`kDS1AttrRecordImage` The record image attribute; clients can use this attribute to force the directory service to generate a binary image of the record and all of its attributes.

`kDSNAttrPlugInInfo` The plugin-information attribute. Clients use this attribute to get information about a Directory Services plug-in, such

	as its version, signature, “about” information, and credits. Multiple values are allowed.
`kDSNAttrRecordName`	The record name attribute. Multiple values are allowed and form a list of names and keys for this record.
`kDSNAttrSchema`	The schema attribute. Multiple values are allowed and form a list of attribute types.
`kDSNAttrRecordType`	The record type attribute. Multiple values are allowed
`kDSNAttrAuthMethod`	The authentication method attribute. Multiple values are allowed.
`kDSNAttrSetPasswdMethod`	The password-setting method attribute. Multiple values are allowed.
`kDSNAttrGroup`	The group attribute. Multiple values are allowed and form a list of group records.
`kDSNAttrMember`	The member attribute. Multiple values are allowed and form a list of member records.
`kDSNAttrURL`	The URL attribute. Multiple values are allowed.
`kDSNAttrMIME`	The MIME attribute. Data stored in this attribute type must be a fully qualified MIME type. Multiple values are allowed.
`kDSNAttrHTML`	The HTML attribute. Multiple values are allowed.
`kDSNAttrNBPEntry`	The Networking Binding Protocol (NBP) attribute. Multiple values are allowed.
`kDSNAttrDNSName`	The Domain Name System (DNS) attribute. Multiple values are allowed
`kDSNAttrIPAddress`	The IP address attribute. Multiple values are allowed
`kDSNAttrPGPPublicKey`	The Pretty Good Privacy (PGP) public key attribute. Multiple values are allowed
`kDSNAttrEMailAddress`	The Email address attribute. Multiple values are allowed
`kDSNAttrPhoneNumber`	The phone number attribute. Multiple values are allowed
`kDSNAttrPostalAddress`	The postal address attribute. Multiple values are allowed

Pattern-Matching Constants

The `tPatternDirMatch` enumeration defines constants for use with Directory Services functions that look for pattern matches. A directory service is not required to support all types of pattern matching.

```
typedef enum
{
    eDSNoMatch1         =   0x0000,
    eDSAnyMatch         =   0x0001,
    eDSBeginAppleReserve1=  0x0002,
    eDSEndAppleReserve1 =   0x1fff,
    eDSExact            =   0x2001,
    eDSStartsWith       =   0x2002,
    eDSEndsWith         =   0x2003,
    eDSContains         =   0x2004,
    eDSLessThan         =   0x2005,
    eDSGreaterThan      =   0x2006,
    eDSLessEqual        =   0x2007,
    eDSGreaterEqual     =   0x2008,
    eDSWildCardPattern  =   0x2009,
    eDSRegularExpression=   0x200A,
    eDSiExact           =   0x2101,
    eDSiStartsWith      =   0x2102,
    eDSiEndsWith        =   0x2103,
    eDSiContains        =   0x2104,
    eDSiLessThan        =   0x2105,
    eDSiGreaterThan     =   0x2106,
    eDSiLessEqual       =   0x2107,
    eDSiGreaterEqual    =   0x2108,
    eDSiWildCardPattern =   0x2109,
    eDSiRegularExpression=  0x210A,
    eDSLocalNodeNames   =   0x2200,
    eDSSearchNodeName   =   0x2201,
    dDSBeginPlugInCustom=   0x3000,
    eDSEndPlugInCustom  =   0x4fff,
    eDSBeginAppleReserve2=  0x5000,
    eDSEndAppleReserve2 =   0xfffe,
    eDSNoMatch2         =   0xffff
} tDirPatternMatch;
```

Constant descriptions

`eDSNoMatch1`

`eDSAnyMatch` Matches any value (case sensitive).

`eDSBeginAppleReserve1` Beginning of a range of values reserved for use by Apple Computer, Inc.

eDSEndAppleReserve1	End of a range of values reserved for use by Apple Computer, Inc.
eDSExact	Matches the specified value exactly (case sensitive).
eDSStartsWith	Matches values that start with the specified value (case sensitive).
eDSEndsWith	Matches values that end with the specified value (case sensitive).
eDSContains	Matches values that contain the specified value (case sensitive).
eDSLessThan	Matches values that are less than the specified value (case sensitive).
eDSGreaterThan	Matches values that are greater than the specifie value (case sensitive).
eDSLessEqual	Matches values that are less than or equal to the specified value (case sensitive).
eDSGreaterEqual	Matches values that are greater than or equal to the specified value (case sensitive).
eDSWildCardPattern	Matches values using the specified wild card pattern (case sensitive).
eDSRegularExpression	Matches values using the specified regular expression (case sensitive).
eDSiExact	Matches the specified value exactly (case insensitive).
eDSiStartsWith	Matches values that start with the specified value (case insensitive).
eDSiEndsWith	Matches values that end with the specified value (case insensitive).
eDSiContains	Matches values that contain the specified value (case insensitive).
eDSiLessThan	Matches values that are less than the specified value (case insensitive).
eDSiGreaterThan	Matches values that are greater than the specifie value (case insensitive).
eDSiLessEqual	Matches values that are less than or equal to the specified value (case insensitive).
eDSiGreaterEqual	Matches values that are greater than or equal to the specified value (case insensitive).

`eDSiWildCardPattern` Matches values using the specified wild card pattern (case insensitive).

`eDSiRegularExpression` Matches values using the specified regular expression (case insensitive).

`eDSLocalNodeNames` Matches the local node name.

`eDSSearchNodeName` Matches the search node.

`dDSBeginPlugInCustom` Beginning of a range of values reserved for use by Directory Services plug-ins.

`eDSEndPluginCustom` End of a range of values reserved for use by Directory Services plug-ins.

`eDSBeginAppleReserve2` Beginning of a range of values reserved for use by Apple Computer, Inc.

`eDSEndAppleReserve2` End of a range of values reserved for use by Apple Computer, Inc.

`eDSNoMatch2`

Result Codes

The result codes specific to Directory Services are listed here. Note that some errors, such as system errors, do not appear in this list.

`eDSNoErr`	0	No error
`eDSOpenFailed`	–14000	Attempt to open a Directory Services session failed
`eDSCloseFailed`	–14001	Attempt to close a Directory Services session failed
`eDSOpenNodeFailed`	–14002	Attempt to open a directory node failed
`eDSBadDirReferences`	–14003	Invalid directory reference
`eDSNullRecordReference`	–14004	Attempt to perform an operation failed because the record reference is null
`eDSMaxSessionsOpen`	–14005	The session limit has bee reached
`eDSCannotAccessSession`	–14006	The specified session is not valid
`eDSDirSrvNotOpen`	–14007	A Directory Services session has not been opened
`eDSNodeNotFound`	–14008	The specified node could not be found

eDSUnknownNodeName	–14009	The specified node name is unknown
eDSRegisterCustomFailed	–14010	Registration of a ustom routine failed
eDSGetCustomFailed	–14011	Unable to get a custom routine
eDSUnRegisterFailed	–14012	Deregistration of a custom routine failed
eDSAllocationFailed	–14050	The requested data type could not be allocated
eDSDeAllocateFailed	–14051	The requested deallocation failed.
eDSCustomBlockFailed	–14052	A custom thread block routine failed
eDSCustomUnblockFailed	–14053	A custom thread unblock routine failed
eDSCustomYieldFailed	–14054	Custom yield routine failed
eDSCorruptBuffer	–14060	A buffer provided as a parameter to a Directory Services function has been corrupted
eDSInvalidIndex	–14061	The specified index is invalid
eDSIndexOutOfRange	–14062	The specified index could not be found
eDSIndexNotFound	–14063	The specified index could not be found
eDSCorruptRecEntryData	–14065	The data in a record entry structure is invalid
eDSRefSpaceFull	–14069	No space is available to accommodate the request
eDSRefTableAllocationError	–14070	The reference could not be allocated
eDSInvalidReference	–14071	The reference is invalid
eDSInvalidRefType	–14072	The reference type is invalid
eDSInvalidDirRef	–14073	Invalid Directory Services reference
eDSInvalidNodeRef	–14074	Invalid node reference
eDSInvalidRecordRef	–14075	Invalid record reference
eDSInvalidAttrListRef	–14076	Invalid attribute list reference
eDSInvalidAttrValueRef	–14077	Invalid attribute value reference
eDSInvalidContinueData	–14078	Invalid continue data value
eDSInvalidBuffFormat	–14079	The format of a buffer is invalid
eDSAuthFailed	–14090	Authentication failed
eDSAuthMethodNotSupported	–14091	The specified authentication is not supported
eDSAuthRespBufTooSmall	–14092	The provided response buffer is too small for the response data

eDSAuthParameterErr	–14093	An authentication parameter is invalid
eDSAuthInBuffFormatError	–14094	A format of a buffer provided for authentication is not correct
eDSAuthNoSuchEntity	–14095	
eDSAuthBadPassword	–14096	The specified password is invalid
eDSAuthContinueDataBad	–14097	The specified continuation data is invalid
eDSAuthUnknownUser	–14098	The specified user name does not exist
eDSAuthInvalidUserName	–14099	The specified user name is invalid
eDSAuthCannotRecoverPassword	–14100	The password could not be obtained
eDSAuthFailedClearTextOnly	–14101	Clear-text authentication failed
eDSAuthNoAuthServerFound	–14102	An authentication server could not be found
eDSAuthServerError	–14103	The authetication server reported an error
eDSInvalidContext	–14104	The provided context data is invalid
eDSBadContextData	–14105	The provided context data is invalid
eDSPermissionError	–14120	The user is not authorized to perform this operation
eDSReadOnly	–14121	The user is not authorized to make modifications
eDSInvalidDomain	–14122	The specified domain is invalid
eNetInfo	–14123	A NetInfo error occurred
eDSInvalidRecordType	–14130	Invalid record type
eDSInvalidAttributeType	–14131	Invalid attribute type
eDSInvalidRecordName	–14133	The record name is invalid
eDSAttributeNotFound	–14134	The requested attribute could not be found
eDSRecordAlreadyExists	–14135	A record of the specifid name and type already exists
eDSRecordNotFound	–14136	The specified record could not be found
eDSNullParameter	–14200	A required parameter is NULL
eDSNullDataBuff	–14201	A required data buffer parameter is NULL
eDSNullNodeName	–14202	The node name is NULL
eDSNullRecEntryPtr	–14203	The record entry pointer is NULL
eDSNullRecName	–14204	The record name is NULL
eDSNullRedNameList	–14205	The list of record names is NULL
eDSNullRecType	–14206	The record type is NULL

eDSNullRecTypeList	–14207	The list of record types is NULL
eDSNullAttribute	–14208	The attribute is NULL
eDSNullAttributeAccess	–14209	The attribute's access flags are NULL
eDSNullAttributeValue	–14210	The attribute's value is NULL
eDSNullAttributeType	–14211	The attribute's type is NULL
eDSNullAttributeTypeList	–14212	The list of attribute types is NULL
eDSNullAttributeControlPtr	–14213	
eDSNullDataList	–14214	The data list is NULL
eDSNullDirNodeTypeList	–14215	The list of directory node types is NULL
eDSNullAuthMethod	–14216	The authentication method is NULL
eDSNullAuthStepData	–14217	The authentication step data is NULL
eDSNullAuthStepDataResp	–14218	The authentication step response data is NULL
eDSNullNodeInfoTypeList	–14219	The list of node type information is NULL
eDSNullPatternMatch	–14220	No match was found
eDSNullNodeNamePattern	–14221	The node name parameter that was specified as a pattern was NULL
eDSEmptyParameter	–14230	A required parameter was empty
eDSEmptyBuffer	–14231	A buffer that shouldhave contained data was empty
eDSEmptyNodeName	–14232	A parameter that should have contained a node name was empty
eDSEmptyRecordName	–14233	A parameter that should have contained a record name was empty
eDSEmptyRecordNameList	–14234	A parameter that shouldhave contained a list of recordnames was empty
eDSEmptyRecordType	–14235	A parameter that should have contained a record type was empty
eDSEmptyRecordTypeList	–14236	A parameter that should have contained a list of record types was empty
eDSEmptyRecordEntry	–14237	A parameter that shouldhave contained a record entry was empty
eDSEmptyPatternMatch	–14238	A parameter that should have contained a pattern to match was empty.
eDSEmptyNodeNamePattern	–14239	
eDSEmptyAttribute	–14240	A parameter that should have contained an attribute was empty

eDSEmptyAttributeType	–14241	A parameter that should have contained an attribute type was empty
eDSEmptyAttributeTypeList	–14242	A parameter that should have contained a list of attribute types was empty
eDSEmptyAttributeValue	–14243	A parameter that should have contained an attribute value was empty
eDSEmptyDataList	–14244	A parameter that should have contained a data list was empty
eDSEmptyNodeInfoTypeList	–14245	A parameter that should have contained a list of node information types was empty
eDSEmptyAuthMethod	–14246	A parameter that should have contained an authentication method was empty
eDSEmptyAuthStepData	–14247	A parameter that shouldhave contained information required for the next step in the authentication process was empty
eDSEmptyAuthStepDataResp	–14248	A parameter that should have contained the response from a previous authentication call was empty
eDSEmptyPattern2Match	–14249	A parameter that should have contain the pattern to match was empty
eDSBufferTooSmall	–14260	A buffer that was supplied as a parameter is too small to contain the results of the call.
eDSUnknownMatchType	–14261	The pattern to match was unknown
eDSUnSupportedMatchType	–14262	The pattern to match is not supported
eDSInvalDataList	–14263	The data list that was provided as a parameter is invalid
eDSAttrListError	–14264	An error occurred for an attribute list
eServerNotRunning	–14270	Directory Services is not running
eUnknownAPICall	–14271	The specific function is not known
eUnknownServerError	–14272	An unknown server error occurred
eUnknownPlugIn	–14273	An unknown plug-in error occurred
ePlugInDataError	–14274	A plug-in data error occurred

ePlugInNotFound	–14275	The requested plug-in could not be found.
ePlugInError	–14276	A plug-in error occurred
ePlugInInitError	–14277	A plug-in initialization error occurred
ePlugInNotActive	–14278	The requested plug-in is loaded but is not active
ePlugInFailedToInitialize	–14279	The plug-in could not initialize itself
ePlugInCallTimedOut	–14280	An attempt to communicate with a plug-in failed
eNoSearchNodesFound	–14290	No search node could be found
eNotHandledByThisNode	–14291	
eIPCSendError	–14300	
eIPCReceiveError	–14331	
eServerReplyError	–14332	
ePluginHandlerNotLoaded	–14400	
eNoPluginsLoaded	–14404	No Directory Services plug-ins are loaded
ePluginAlreadyLoaded	–14406	The requested plug-in is already loaded
ePluginPrefixNotFound	–14404	
eDuplicatePluginPrefix	–14408	
eNoPluginFactoriesFound	–14410	
eCFMGetFileSysRepErr	–14450	
eCFPlugInGetBundleErr	–14452	
eCFBndleGetInfoDictErr	–14454	
eCFDictGetValueErr	–14456	
eDSServerTimeout	–14458	
eDSContinue	–14470	
eDSInvalidHandle	–14472	
eDSSendFailed	–14473	
eDSReceiveFailed	–14474	
eDSBadPacket	–14475	A corrupt packet was received
eDSInvalidTag	–14476	The specified tag is invalid
eDSInvalidSession	–14477	The specified Directory Service reference is not valid
eDSInvalidName	–14478	The specified name is not valid
eDSUserUnknown	–14479	The specified user is not a valid user
eDSUnrecoverablePassword	–14480	The password could not be recovered
eDSAuthenticationFailed	–14481	The authentication failed
eDSBogusServer	–14482	
eDSOperationFailed	–14483	The operation failed

eDSNotAuthorized	–14484	The user is not authorized to perform the request operation
eDSNetInfoError	–14485	A NetInfo error occurred
eDSContactMaster	–14486	
eDSServiceUnavailable	–14487	The request service is not available
eMemoryError	–14900	A memory error occurred
eMemoryAllocError	–14901	A memory allocation error occurred
eServerError	–14902	A server error occurred
eUndefinedError	–14987	An undefined error occurred
eNotYetImplemented	–14988	The specifid function is not implemented yet.

User Attributes

This appendix lists user attributes and describes their characteristics.

Primary Record Name Attribute

Required: Yes
Editable by administrator: Yes
Maximum size: 128 bytes
Multiple values allowed: No
Case-sensitive: No
Search: Substring
Read access: Everyone
Write access: Self, administrator
Format: XML object. An array of Unicode characters.

Record Password Attribute

Required: Yes
Editable by administrator: Yes
Maximum size: 128 bytes
Multiple values allowed: No
Case-sensitive: Yes
Search:
Read access: Authetication agent same CPU
Write access: Self, administrator
Format: Proprietary non-public format. Location, format and encoding are left to the discretion of the authentication service.

Internet Alias Attribute

Required: No
Editable by administrator: Yes
Maximum size: 128 bytes
Multiple values allowed: No
Case-sensitive: No
Search: Substring
Read access: Everyone
Write access: Self, administrator
Format: XML object. An array of 7-bit characters. No spaces, limited symbols.

Record Alias Attribute

Required: No
Editable by administrator: Yes
Maximum size: 128 bytes each, 16 entries
Multiple values allowed: Yes
Case-sensitive: No
Search: Substring
Read access: Everyone
Write access: Self, administrator
Format: XML objects. Array of Unicode characters.

Record ID Attribute

Required: Yes
Editable by administrator: Yes
Maximum size: 64 bytes of XML text
Multiple values allowed: No
Case-sensitive: Not applicable
Search: Exact match
Read access: Everyone
Write access: Administrator
Format: XML object. Legal 32-bit unsigned numeric value, encoded as an 8-digit hexidecimal / ASCII string.

Primary Group ID Attribute

Required: Yes
Editable by administrator: Yes
Maximum size: 512 bytes
Multiple values allowed: No
Case-sensitive: Not applicable
Search:
Read access: Everyone
Write access: Administrator
Format: An XML object having the following format:

```
<node_path> <group_ID> <group_primary_name>,
```

where

```
<node_path> = <node_count> <array_of_Unicode_characters>
... <array_of_Unicode_characters>
<node_count> = <hex_ASCII_number>
<group_ID> = <hex_ASCII_number>
<group_primary_name> = <array_of_Unicode_characters>
<array_of_Unicode_characters> = <hex_ASCII_number>
```

<hex_ASCII_number> = “0x” + contiguous series of ASCII values “a” through “f” and “0” through “9”>

Group Membership Attribute

Required: No
Editable by administrator: Yes
Maximum size: 512 bytes per entry, 128 entries
Multiple values allowed: Yes
Case-sensitive: No
Search:
Read access: Everyone
Write access: Administrator
Format: An XML object having the following format:

<node_path> <group_ID> <group_primary_name>,
where
<node_path> = <node_count> <array_of_Unicode_characters>
<node_count> = <hex_ASCII_number>
<group_ID> = <hex_ASCII_number>
<group_primary_name> = <array_of_Unicode_characters>
<array_of_Unicode_characters> = <hex_ASCII_number>
<hex_ASCII_number> = “0x” + contiguous series of ASCII values “a” through “f” and “0” through “9”>

Home Directory Attribute

Required: No
Editable by administrator: Yes
Maximum size: 1024 bytes
Multiple values allowed: No
Case-sensitive: No
Search:
Read access: Everyone
Write access: Administrator
Format: An XML object having the following format:

<file_protocol> <transport_type> <network_address>
<share_point_name> <full_path>
where
<file_protocol> = “AFP” | “SMB” | “NFS” | “FTP” | “WEBDAV”
<transport_type> = “AT” | “TCP/IP” | “IPX” | “X25”
<network_address> = “<address_type>:<address_value>
<address_type> = “NBP” | “IP” | “DNS” | “HARDWARE”
<address_value> = <array_of_Unicode_characters>
<sharepoint_name> = <array_of_Unicode_characters>

```
<full_path> = <path_item_count> <array_of_Unicode_characters>
... <array_of_Unicode_characters>
<array_of_Unicode_characters> = <hex_ASCII_number>
<Unicode_character> ... <Unicode_character>
<hex_ASCII_number> = "0x" + contiguous series of ASCII values
"a" through "f" and "0" through "9">
```

Mail Account Attribute

Required: No
Editable by administrator: Yes
Maximum size: Internal specification
Multiple values allowed: No
Case-sensitive: Not applicable
Search:
Read access: Everyone
Write access: Administrator
Format: XML object. Mail server account options

Email Address Attribute

Required: No
Editable by administrator: Yes
Maximum size: 2048 bytes per entry, 16 entries
Multiple values allowed: Yes
Case-sensitive: No
Search:
Read access: Everyone
Write access: Self, administrator
Format: XML object. Standard mailto:URL

Comment Attribute

Required: No
Editable by administrator: Yes
Maximum size: 2048 bytes
Multiple values allowed: No
Case-sensitive: No
Search:
Read access: Everyone
Write access: Administrator
Format: XML object. Standard HTML 2.0.

RARA Attribute

Required: No
Editable by administrator: Yes

Maximum size: (TBD)
Multiple values allowed: No
Case-sensitive: No
Search:
Read access: Everyone
Write access: Administrator
Format: (TBD).

Generated Unique ID Attribute

Required: Yes
Editable by administrator: No
Maximum size: 256 bits
Multiple values allowed: No
Case-sensitive: Not applicable
Search:
Read access: Everyone
Write access: Write once on create
Format: XML object. Persistent, hexadecimal, numeric ASCII value up to 256 bits (64 hexadecimal digits); high entropy value.

Telephone Number Attribute

Required: No
Editable by administrator: Yes
Maximum size: 64 bytes per entry; 16 entries
Multiple values allowed: Yes
Case-sensitive: No
Search:
Read access: Everyone
Write access: Self, administrator
Format: XML object. Use external standards.

Administrator Attribute

Required: No
Editable by administrator: Yes
Maximum size: Not applicable
Multiple values allowed: Not applicable
Case-sensitive: Not applicable
Search: Not applicable
Read access: Everyone
Write access: Administrator
Format: Administrator access will be dictated by membership in a "well-known" group.

Password Aging Policy Attribute

Required: Not applicable
Editable by administrator: Not applicable
Maximum size: Not applicable
Multiple values allowed: Not applicable
Case-sensitive: Not applicable
Search: Not applicable
Read access: Not applicable
Write access: Not applicable
Format: Not applicable

Index