I N S I D E   M A C I N T O S H

Network Setup

Contents

Figures, Tables, and Listings

About This Manual

This manual describes Network Setup, which is a programming interface that allows you to manipulate the contents of the Network Setup database. The Network Setup database contains settings for all of the network protocols installed on the system. Using Network Setup, you can programmatically modify any network setting that the user can see in the various networking control panels.

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 Network Setup programming interface:

- *Inside AppleTalk,* Second Edition.
- *Inside Macintosh: Networking with Open Transport.*
- *Open Transport Advanced Client Programming,* available at `http://developer.apple.com/macos/opentransport/OTAdvancedClientProg/OTAdvancedClientProg.html`

About Network Setup

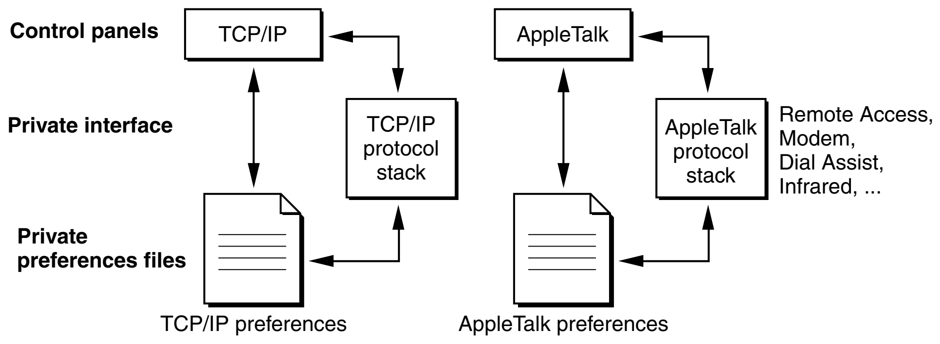
Network Setup is a system service that allows you to manipulate network configurations. You can use Network Setup to read, create, modify, and delete network configurations. Any option that is accessible to the user through the network control panels provided by Apple is also available to you through the Network Setup programming interface.

This chapter describes the overall Network Setup architecture and introduces the terminology needed to understand how to use Network Setup. It assumes that you are familiar with the existing network control panels provided by Apple (for example, the TCP/IP control panel) from a user's perspective, especially the configurations window used to select, duplicate, and rename network configurations.

Network Setup Architecture

Prior to the introduction of Network Setup, each network protocol stack used its own private mechanism to store preferences and make those preferences active. Network preferences were stored as resources in files in the Preferences folder. Figure 1-1 shows the overall network configuration architecture prior to the introduction of Network Setup.

Figure 1-1 Network configuration prior to Network Setup



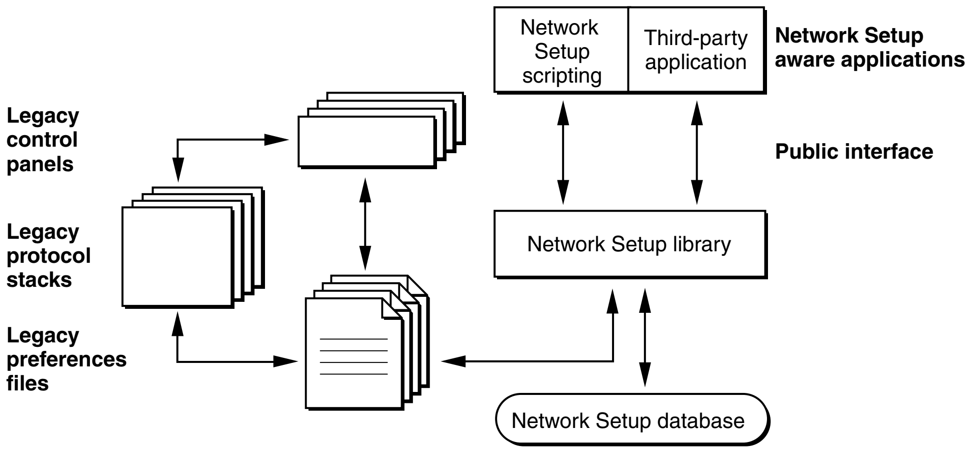
The architecture shown in Figure 1-1 had a number of drawbacks:

- There was a control panel for each protocol type, leading to an unnecessary proliferation of control panels.
- There was no programming interface for changing network settings. With the explosion of interest in networking prompted by the rise of the Internet, this proved to be a problem. Internet setup programs, whether provided by Apple or by third parties, were required to reverse engineer the network preferences file format. After changing the files "underneath" the protocol stacks, these programs had to force the protocol stack to read the new preferences through a variety of unsupported means.
- The dependence of third-party applications on the preferences file format and private interfaces to the protocol stack made it difficult for Apple to ship modern network features, such as TCP/IP multihoming, and to support the multiple users feature in Mac OS 9.
- Resource files are susceptible to corruption when the system crashes.

Network Setup was designed to eliminate these problems by giving developers, both inside and outside of Apple, a programming interface to modify network preferences without relying on internal implementation details of the individual protocol stacks.

Network Setup is being introduced in two stages. The architecture of the first stage (Mac OS 8.5 to the present day) is shown in Figure 1-2.

Figure 1-2 Network Setup in Mac OS 8.5 through the present

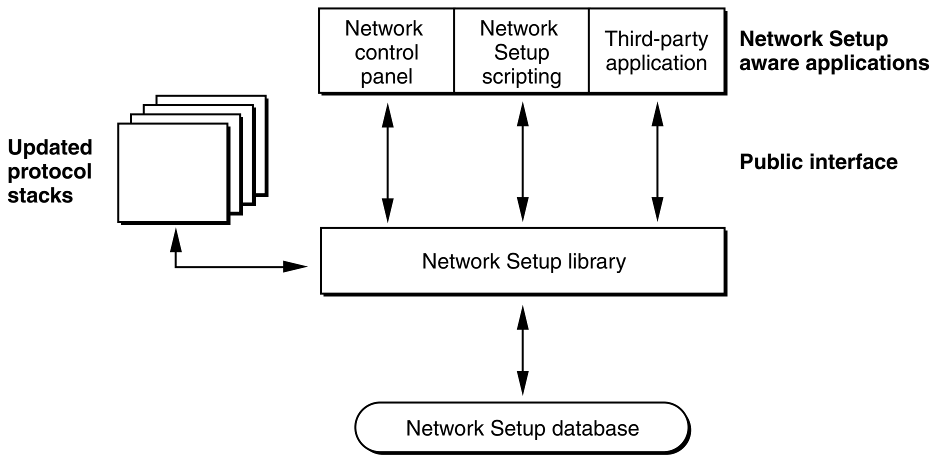


The following key points are to be taken from Figure 1-2:

- The Network Setup library provides a standard programming interface for manipulating network configurations stored in the Network Setup database. The database is designed to store network preferences reliably even if the system crashes while preferences are being modified.
- The Network Setup library provides automatic synchronization between the database and legacy preference files. Synchronization allows existing software with dependencies on the format of those files (such as third-party Internet setup software, Apple control panels, and protocol stacks) to continue working in the new environment.
- Third-party developers are encouraged to migrate to the Network Setup programming interface, but in so doing, their existing applications in the field will not break.
- Network Setup scripting is a bridge between the Network Setup programming interface and AppleScript. It allows script developers to manipulate network configurations through a standard AppleScript object model interface.

The primary disadvantage of the current Network Setup architecture is that the synchronization between the legacy preferences files and the Network Setup database is a time consuming operation. Consequently, Apple intends to remove support for legacy preferences files as soon as possible. Figure 1-3 shows the future Network Setup architecture.

Figure 1-3 Future Network Setup architecture

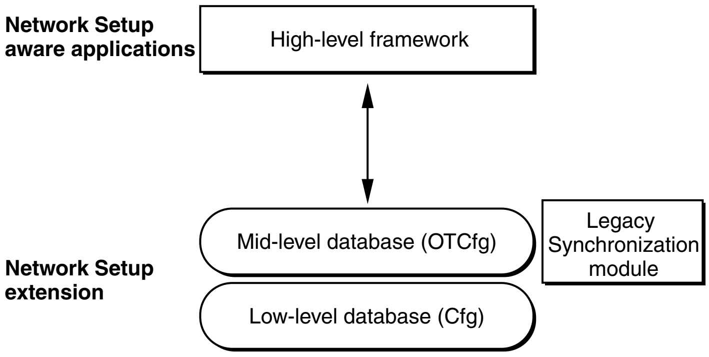


In the future Network Setup architecture, all developers, applications that manipulate network preferences will be required to use the Network Setup programming interface. If you have an application that manipulates legacy preferences files directly, to guarantee future compatibility you must update it to use the Network Setup programming interface.

Inside the Network Setup Library

Figure 1-4 shows the structure of the Network Setup library itself and its relationship to the applications that call it. This structure is mostly irrelevant to programmers who call the programming interface — Network Setup acts like a "black box"—- but it helps to explain how Network Setup works.

Figure 1-4 Structure of the Network Setup Library


Network Setup aware applications
High-level framework
Mid-level database (OTCfg)
Legacy Synchronization module
Network Setup extension
Low-level database (Cfg)

As shown in Figure 1-4, the Network Setup library is divided into four key components:

- The low-level database, which is an internal component of the Network Setup Extension file. The low-level database contains the core database manipulation engine. It knows nothing about networking — it just moves bits around. The low-level database is not visible to developers except insofar as its prefix ("Cfg") is used by some Network Setup identifiers.
- The mid-level database, which is the actual programming interface exported to developers. Its routine names start with "OTCfg". The mid-level database passes most requests directly to the low-level database, which actually executes the request and manipulates the database. The mid-level database also interfaces with the legacy synchronization module.
- The legacy synchronization module, which in combination with the mid-level database, ensures that the database is synchronized with the legacy preferences files. This module will be removed in a future version of Network Setup. See "Legacy Synchronization Algorithm" (page 22) for more information about legacy file synchronization.
- Most users of the Network Setup programming interface use a high-level framework to assist them in their task. Apple software uses an Apple-internal C++ framework for this. This framework is statically linked into software like the Network Setup Scripting application. Third-party developers commonly use the MoreNetworkSetup framework, available as sample code.

Network Setup Database Fundamentals

This section describes the fundamental structure of and operations on the Network Setup database.

Database Structure

The Network Setup database consists of multiple areas. There are two types of areas: **named areas** store preferences, while temporary areas are used as part of the preference modification process. The system currently uses a single named area, known as the **default area** (sometimes referred to as the **current area**) to store all network preferences. While it is possible to create and manipulate other named areas within the database, doing so does not affect any network settings. Areas are identified by a unique **area ID**.

Each area contains a number of **entities** having the following properties:

- **entity reference.** An entity reference uniquely identifies an entity. The entity reference contains an area ID, which identifies the area in which the entity resides.
- **entity name.** A user-visible name for the entity that need not be unique.
- **entity class** and **type.** These values, both of type `OSType`, determine the type of data contained within an entity. There are three entity classes:
    - **network connection entity.** A network connection entity contains information about a single instance of a network protocol on a port. Typically there is one active network connection entity per protocol stack, but on a multihomed computer there can be more. The entity type for an network connection entity indicates the network protocol of the connection.
    - **global protocol entity.** A global protocol entity contains configuration for a protocol stack on a computer. There is only one active global protocol entity for each protocol stack. The entity type for a global protocol entity indicates the network protocol whose configuration it contains.
    - **set entity.** A set entity groups global protocol and network connection entities into a set. The set entity contains entity references to each entity in the set. An area can contain multiple set entities, but there is one and only

one **active set entity**. The entities referenced by the active set entity comprise the active network preferences. All set entities have the same type.

- **icon.** An entity can include a reference to a custom icon. The custom icon is not currently used, but may be used by future system software to display a visual representation of the entity.

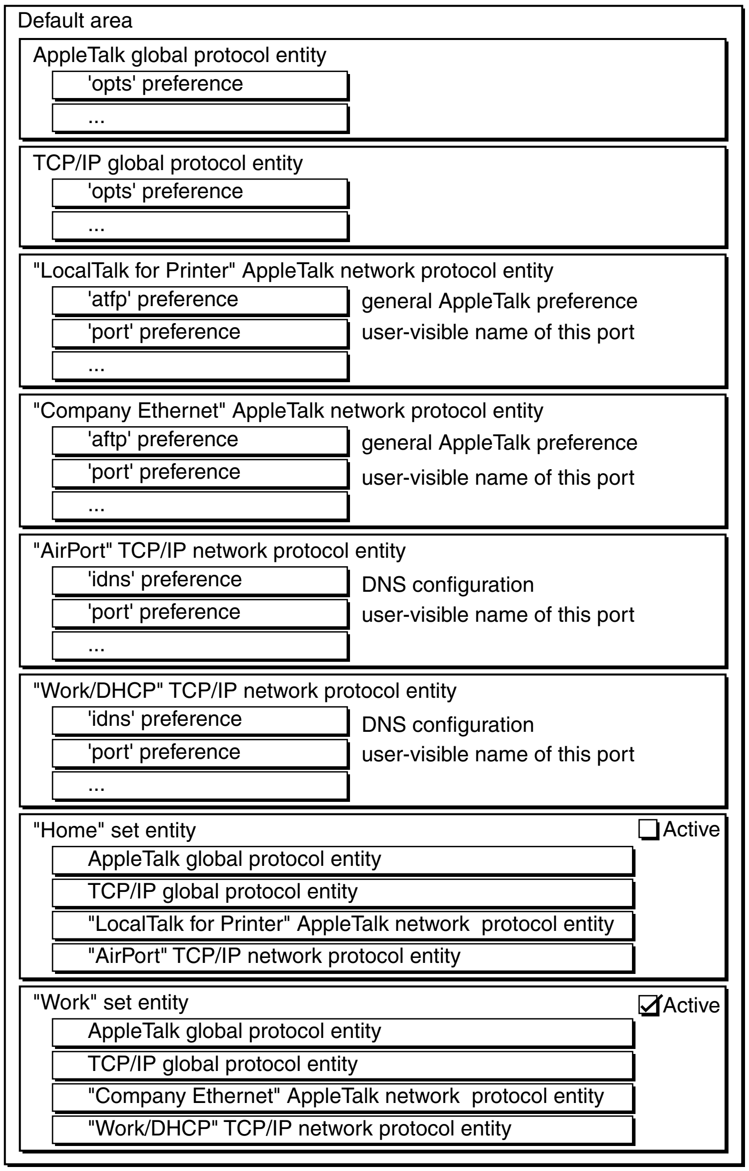
Within each entity there are zero or more **preferences**, distinguished by a **preference type** (an `OSType`). A preference is the atomic unit of data in the database. When you read or write data, you do so one preference at a time. Typically the data for a preference is protocol-dependent. Its format is determined by the entity class and type and by the preference type itself. To read or write a preference meaningfully, you must know the format of the preference data. The reference section of this document describes the format of every preference used by the Apple protocol stacks. In most cases, this description includes a C structure that mirrors the structure of the preference itself.

Note
For most preferences, the data format is the same as for the equivalent resource in the legacy preference files. If you are familiar with the legacy file format, you should be able to easily understand the preference data format. See "Legacy Issues" (page 21) for more information on how Network Setup synchronizes the database with the legacy preferences files.

Database Structure Example

Figure 1-5 shows an example of how the Network Setup database might be structured on a particular computer.

Figure 1-5 Sample organization of the default area


Default area
AppleTalk global protocol entity
'opts' preference
...
TCP/IP global protocol entity
'opts' preference
...
"LocalTalk for Printer" AppleTalk network protocol entity
'atfp' preference
general AppleTalk preference
'port' preference
user-visible name of this port
...
"Company Ethernet" AppleTalk network protocol entity
'aftp' preference
general AppleTalk preference
'port' preference
user-visible name of this port
...
"AirPort" TCP/IP network protocol entity
'idns' preference
DNS configuration
'port' preference
user-visible name of this port
...
"Work/DHCP" TCP/IP network protocol entity
'idns' preference
DNS configuration
'port' preference
user-visible name of this port
...
"Home" set entity
Active
AppleTalk global protocol entity
TCP/IP global protocol entity
"LocalTalk for Printer" AppleTalk network protocol entity
"AirPort" TCP/IP network protocol entity
"Work" set entity
✓ Active
AppleTalk global protocol entity
TCP/IP global protocol entity
"Company Ethernet" AppleTalk network protocol entity
"Work/DHCP" TCP/IP network protocol entity

For simplicity, this example assumes a computer with two places of operation, home and work, and two protocol stacks, TCP/IP and AppleTalk. Thus, there are four network connection entities:

- "AirPort," a TCP/IP network connection entity that configures a TCP/IP interface to use an AirPort card to access an AirPort Base Station at home.
- "LocalTalk for Printer," an AppleTalk network connection entity that configures an AppleTalk interface to use LocalTalk over the Printer port, to talk to a LocalTalk printer at home.
- "Work/DHCP," a TCP/IP network connection entity which configures a TCP/IP interface to use DHCP over the Ethernet port.
- "Company Ethernet," an AppleTalk network connection entity that configures an AppleTalk interface to use the Ethernet port in a zone that only exists on the "Work" network.

The area also has two global protocol entities, one for TCP/IP and one for AppleTalk. These settings do not need to change between home and work, so there is only one of each.

Finally, the area has two set entities:

- "Home," which references the two global protocol entities and the two home network connection entities: "AirPort" for TCP/IP and "LocalTalk for Printer" for AppleTalk.
- "Work," which references the two global protocol entities but also references two network connection entities: "Work/DHCP" for TCP/IP and "Company Ethernet" for AppleTalk.

The "Work" set entity is marked as active, so the network connection entities that it references are active. When the user moves from work to home, a program (such as the Location Manager) can simply mark the "Work" entity as inactive and the "Home" entity as active and the network configuration will switch accordingly.

Database Operations

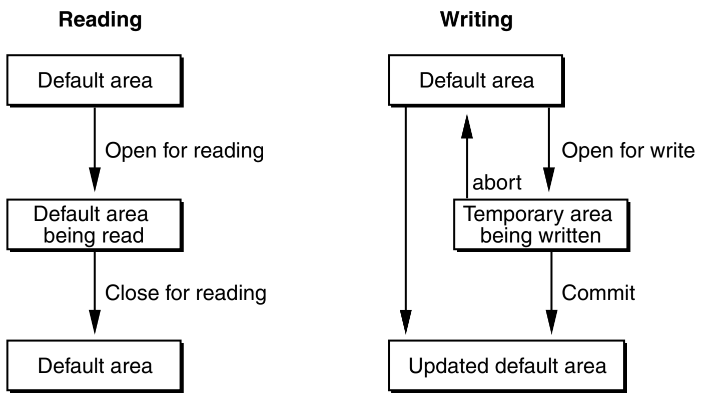
Before reading or writing preferences, an application must open the database. The first step is to create a **database reference**. This reference identifies the calling application to the Network Setup library and is passed to subsequent calls that access the database. After creating the database reference, the process diverges for readers and writers. When an application opens the default area for reading, it reads the area directly. Network Setup simply notes that the area is open for synchronization purposes (see the section "Preference Coherency" (page 21)). For writing, the process is somewhat different.

When an application opens an area for writing, Network Setup creates a temporary area that is an exact duplicate of the default area. It then returns the temporary area ID to the application. The application can now change the temporary area without affecting running network services. When the application is done making changes, it commits the changes to the database. Committing changes is an atomic process that overwrites the default area with the contents of the temporary area and causes the protocol stacks to reconfigure themselves for the new configuration.

Alternatively, the writing application can choose to abort the modifications, in response to which Network Setup discards the temporary area and the system continues to use the configuration in the default area.

Figure 1-6 shows this process diagrammatically.

Figure 1-6 Reading and writing the default area



Multiple applications can open the Network Setup database for reading, but only one application at a time can open the database for writing.When an application commits changes to the default area, Network Setup notifies each application that has opened the database for reading that a change has occurred, as explained in the next section, "Preference Coherency."

Preference Coherency

When an application commits changes to the default area, it is important that applications that are reading the database be informed of those changes. For example, an application might be displaying the DHCP client ID preference. If another application changes this preference in the database, it is important that the original application update its display.

Prior to Network Setup 1.0.2, the mechanism by which readers learned of changes was somewhat primitive. When a writing application committed its changes, Network Setup tagged each reading application's database reference with an error. Any subsequent calls using that database connection failed with an error (`kCfgErrDatabaseChanged`). The reading application responded by closing its database reference and opening the database again. It then read the new preferences.

Network Setup 1.0.2 introduces a new, notifier-based mechanism for learning about preference changes. The reading application attaches a **notifier** to the database reference. When changes are committed to the database, Network Setup calls each installed notifier to inform the reading application that a change has occurred. The reading application should respond by re-reading any preferences it has cached.

Legacy Issues

As described in the section "Network Setup Architecture" (page 11), current versions of Network Setup automatically synchronize the legacy preferences files with the database. This synchronization mechanism is transparent to applications calling Network Setup, but there are two issues that you should be aware of:

- Legacy synchronization is slow. Depending on the speed of the computer and the number of entities, a full synchronization can take seconds. You should do all that you can to avoid synchronizations. The best way to avoid synchronizations is to adopt Network Setup and avoid modifying the legacy preferences files directly.
- Legacy preferences files do not support multihoming.

Given that legacy synchronization is slow and that legacy preferences files do not support multihoming, future system software will not support legacy synchronization.

Legacy Synchronization Algorithm

Network Setup synchronizes the database and the legacy preferences files at the following times:

- When the database is opened. Network Setup checks the modification dates of each legacy preferences file against modification dates stored in the database. If the dates are different, Network Setup assumes that some application has changed one or more legacy preferences files and imports preferences from the modified files.
- When changes are committed to the database. Network Setup determines whether the committed changes affect a legacy preferences file. If they do, Network Setup exports the database to the legacy preferences file and records the modification date of the legacy preferences file in the database.

The legacy import mechanism makes good use of the structure of the legacy preferences files. Most preferences files are resource files having the following attributes:

- A resource having a well known resource type (`'cnam'`).
- All resources with a resource ID of a `'cnam'` resource belong to that configuration. The contents of these resources are the preferences for that configuration.
- There is one fixed resource whose type is `'ccfg'` and whose ID is 1 that contains the resource ID of the active configuration.
- Any resources with IDs other than those used for configurations are global preferences.

When importing a legacy preferences file, Network Setup creates an entity for each `'cnam'` resource and, for each resource with the same ID as the `'cnam'` resource, creates a preference in the entity whose preference type is the resource ID and whose data is the resource data. If the `'ccfg'` resource indicates that the configuration is active, Network Setup places the entity in the active set.

The legacy export process is similar to the legacy import mechanism. For each network connection entity of a particular type, Network Setup creates a `'cnam'` resource with a unique ID in the legacy file. Then, for each preference in the entity, it creates a resource containing the preference data with the resource type matching the preference type and the resource ID the same as the `'cnam'` resource.

Network Setup uses a number of private preferences to ensure a reliable round trip conversion between legacy preferences files and the database. The preference types are described in "Common Preference Types" (page 106), but your application should not depend on their presence, their content, or their semantics.

If you find undocumented preferences (such as a preference of type `'vers'`) in a global protocol entity, do not be concerned. Network Setup itself does not actually look at the data as it imports from and exports to legacy preferences files.

Network Setup Version History

Table 1-1 summarizes the different versions of Network Setup, their features, and their release vehicles.

Table 1-1 Network Setup versions

Version	Mac OS version	New features
1.0	Mac OS 8.5	
1.0.1	Not released	`OTCfgGetAreaName`
1.0.2	Mac OS 8.6	`OTCfgInstallNotifier`, `OTCfgRemoveNotifier`
1.1	Not released	`OTCfgEncrypt`, `OTCfgDecrypt`
1.1.1	Mac OS 9.0	

There is no easy way to determine the version of Network Setup installed on a system. The best way to test for the presence of a specific Network Setup API enhancement is to weak link to its symbol, as described in TN 1083, "Weak Linking to a Code Fragment Manager-based Shared Library."

Using Network Setup

This chapter explains how to use the Network Setup programming interface to read and write network preferences. It assumes that you are familiar with basic Network Setup concepts. If not, you should read Chapter 1, "About Network Setup,"for important background material. This chapter concentrates on practical examples of coding with Network Setup.

Opening and Closing the Network Setup Database

This section explains how your application should open the Network Setup database for reading and writing and then discusses how to close the database and, in the case where the database has been opened for writing, either committing or discarding your modifications.

Opening the Database for Reading

The `MyOpenDatabaseForReading` routine shown in Listing 2-1 shows how to open the default (or current) area in the Network Setup database for reading. It starts by calling `OTCfgOpenDatabase`, which returns a database reference (of type `CfgDatabaseRef`) that identifies your application's connection to the database. It then calls `OTCfgGetCurrentArea`, which returns an area identifier (of type `CfgAreaID`) that identifies the default area. Finally, it opens the default area for reading by calling `OTCfgOpenArea`.

The `MyOpenDatabaseForReading` routine returns both the database reference (`dbRef`) and the default area identifier (`readArea`). You must know these values in order to read preferences and eventually close the database.

Listing 2-1 Opening the database for reading

```
static OSStatus MyOpenDatabaseForReading(CfgDatabaseRef *dbRef,
                                    CfgAreaID *readArea)
{
    OSStatus err;
    assert(dbRef    != nil);
    assert(readArea != nil);

    err = OTCfgOpenDatabase(dbRef);
    if (err == noErr) {
        err = OTCfgGetCurrentArea(*dbRef, readArea);
        if (err == noErr) {
            err = OTCfgOpenArea(*dbRef, *readArea);
        }
        if (err != noErr) {
            (void) OTCfgCloseDatabase(dbRef);
        }
    }

    if (err != noErr) {
        *dbRef    = nil;
        *readArea = kInvalidCfgAreaID;
    }

    return err;
}
```

Opening the Database for Writing

The `MyOpenDatabaseForWriting` routine shown in Listing 2-2 shows how to open the default (or current) area in the Network Setup database for writing. The approach is similar to that used for opening the database for reading except that instead of calling `OTCfgOpenArea` to open the area for reading, the routine calls `OTCfgBeginAreaModifications` to open the area for writing.

The `OTCfgBeginAreaModifications` function returns another area identifier that references a writable temporary area. The `MyOpenDatabaseForWriting` routine returns both the original default area identifier and the writable temporary area identifier. You must keep both of these area identifiers because you need them both in order to close the database. You can only make changes to the writable area, but you can read from both the original area and the writable area to access, respectively, the currently active network settings and your proposed changes to the network settings.

Listing 2-2 Opening the database for writing

```
static OSStatus MyOpenDatabaseForWriting(CfgDatabaseRef *dbRef,
                                    CfgAreaID *readArea,
                                    CfgAreaID *writeArea)
{
    OSStatus err;

    assert(dbRef     != nil);
    assert(writeArea != nil);

    err = OTCfgOpenDatabase(dbRef);
    if (err == noErr) {
        err = OTCfgGetCurrentArea(*dbRef, readArea);
        if (err == noErr) {
            err = OTCfgBeginAreaModifications(*dbRef, *readArea,
writeArea);
        }
        if (err != noErr) {
            (void) OTCfgCloseDatabase(dbRef);
        }
    }

    if (err != noErr) {
        *dbRef     = nil;
        *readArea  = kInvalidCfgAreaID;
        *writeArea = kInvalidCfgAreaID;
    }

    return err;
}
```

Closing the Database After Reading

The `MyCloseDatabaseAfterReading` routine shown in Listing 2-3 shows how to close the database after you are done reading from it. The routine simply calls `OTCfgCloseArea` to close the read area and then calls `OTCfgCloseDatabase` to close the database itself. This code discards error results from both of these routines because if the database fails to close there isn't anything your application can do to force it to close, but it does log any errors with the standard C `assert` macro so that you can detect this sort of error during testing.

Listing 2-3 Closing the database after reading

```
static void MyCloseDatabaseAfterReading(CfgDatabaseRef dbRef,
                                    CfgAreaID readArea)
{
    OSStatus junk;

    assert(dbRef    != nil);
    assert(readArea != kInvalidCfgAreaID);

    junk = OTCfgCloseArea(dbRef, readArea);
    assert(junk == noErr);
    junk = OTCfgCloseDatabase(&dbRef);
    assert(junk == noErr);
}
```

Closing the Database After Writing

The `MyCloseDatabaseAfterWriting` routine shown in Listing 2-4 shows how to close the database after you have finished making changes. The first three routine parameters (the database reference, the read area identifier, and the write area identifier) were obtained when the database was opened the database for writing. The fourth parameter, `commit`, indicates whether the changes are to be committed to the database or discarded.

If `commit` is true, the routine calls `OTCfgCommitAreaModifications`, which overwrites the current settings in the read area with the new settings in the write area and notifies the network protocol stacks that their preferences have changed so that they can reconfigure themselves.

If `commit` is false, the routine calls `OTCfgAbortAreaModifications` to discard the changes made in the writable temporary area. The read area is not changed, and the network protocol stacks continue unaffected.

In contrast to `MyCloseDatabaseAfterReading` shown in Listing 2-3, `MyCloseDatabaseAfterWriting` does not always throw away error results. If `OTCfgCommitAreaModifications` returns an error, the `MyCloseDatabaseAfterWriting` routine aborts. Your application may respond to this by calling the same routine again, this time with `commit` set to `false`.

Listing 2-4 Closing the database after writing

```
static OSStatus MyCloseDatabaseAfterWriting(CfgDatabaseRef dbRef,
                                    CfgAreaID readArea,
                                    CfgAreaID writeArea,
                                    Boolean commit)
{
    OSStatus err;
    OSStatus junk;

    assert(dbRef     != nil);
    assert(readArea  != kInvalidCfgAreaID);
    assert(writeArea != kInvalidCfgAreaID);

    if ( commit ) {
        err = OTCfgCommitAreaModifications(dbRef, readArea, writeArea);
    } else {
        junk = OTCfgAbortAreaModifications(dbRef, readArea);
        assert(junk == noErr);
        err = noErr;
    }
if (err == noErr) {
    err = OTCfgCloseDatabase(&dbRef);
    }

return err;

}
```

Working with Entities

Once you have a reference to the database and an area identifier for the default area, the next step is to look for appropriate entities within that area. Regardless of what you want to do to an entity, you must first obtain a reference to it. An entity reference is an opaque data structure that Network Setup uses to uniquely identify each entity within an area.

Typically there are two ways to get the entity reference for an entity within an area.

- You can list all of the entities of a particular class and type and display that list to the user. For example, you might want to list all of the TCP/IP network connection entities so that the user can choose the one to make active. The section "Listing All Entities" (page 30) explains how to do this.
- You can find the currently active entity of a particular class and type. This is useful when you want to read the current network settings. The section "Finding an Active Entity" (page 34) explains how to do this.

You can also create, duplicate, rename, and delete entities. These tasks are easy to do and are not explained in detail in this chapter. See "Network Setup Reference" (page 57) for information about `OTCfgCreateEntity` (page 76), `OTCfgDuplicateEntity` (page 78), `OTCfgSetEntityName` (page 81), and `OTCfgDeleteEntity` (page 78).

Listing All Entities

Listing 2-5 shows the routine `MyGetEntitiesList`, which generates a list of all of the entities within an area of the database. The routine's database reference and area identifier parameters are obtained by opening the database, as explained in "Opening the Database for Reading" (page 25). The `entityClass` and `entityType` parameters specify the entities to list. Some common scenarios include:

- Getting all entities of a specific class and type. Set the `entityClass` and `entityType` parameters to that class and type. For example, to find all TCP/IP network connection entities, supply a class of `kOTCfgClassNetworkConnection` and a type of `kOTCfgTypeTCPv4`. See "Entity Classes and Types" (page 104) for a list of the defined classes and types.

- Getting all entities. Set `entityClass` and `entityType` to the wildcard values `kCfgClassAnyEntity` and `kCfgTypeAnyEntity`, respectively.

The `entityRefs` and `entityInfos` parameters are handles containing an array of elements of type `CfgEntityRef` and `CfgEntityInfo`, respectively. You must create these handles before calling `MyGetEntitiesList`. You can set `entityInfos` to `NULL` if you're not interested in the information returned in that handle. The `MyGetEntitiesList` routine resizes the handles appropriately to hold information about each of the entities that it finds.

The `MyGetEntitiesList` routine calls two key Network Setup functions: `OTCfgGetEntitiesCount` to count the number of entities of the specified class and type and `OTCfgGetEntitiesList` to get the actual entity information. The rest of the `MyGetEntitiesList` routine is just memory management.

Listing 2-5 Finding all entities of a particular class and type

```
static OSStatus MyGetEntitiesList(CfgDatabaseRef dbRef,
                            CfgAreaID area,
                            OSType entityClass,
                            OSType entityType,
                            CfgEntityRef  **entityRefs,
                            CfgEntityInfo **entityInfos)
{
    OSStatus err;
    ItemCount entityCount;
    CfgEntityRef  *paramRefs;
    CfgEntityInfo *paramInfos;
    SInt8 sRefs;
    SInt8 sInfos;

    assert(dbRef != nil);
    assert(area  != kInvalidCfgAreaID);
    assert((entityRefs  != nil) || (entityInfos  != nil));
    assert((entityRefs  == nil) || (*entityRefs  != nil));
    assert((entityInfos == nil) || (*entityInfos != nil));

    err = OTCfgGetEntitiesCount(dbRef, area, entityClass, entityType, &entityCount);
    if ((err == noErr) && (entityRefs != nil)) {
        SetHandleSize( (Handle) entityRefs, entityCount * sizeof(CfgEntityRef) );
        err = MemError();
    }

    if ((err == noErr) && (entityInfos != nil)) {
        SetHandleSize( (Handle) entityInfos, entityCount * sizeof(CfgEntityInfo) );
        err = MemError();
    }

    if (err == noErr) {
        if (entityRefs == nil) {
            paramRefs = nil;
        } else {
            sRefs = HGetState( (Handle) entityRefs  );      assert(MemError() ==
noErr);
            HLock( (Handle) entityRefs );                   assert(MemError() ==
noErr);
            paramRefs = *entityRefs;
        }
        if (entityInfos == nil) {
            paramInfos = nil;
        } else {
            sInfos = HGetState( (Handle) entityInfos );     assert(MemError() ==
noErr);
            HLock( (Handle) entityInfos );                  assert(MemError() ==
noErr);
            paramInfos = *entityInfos;
        }

        err = OTCfgGetEntitiesList(dbRef, area,
                                entityClass, entityType,
                                &entityCount, paramRefs, paramInfos);

        if (entityRefs != nil) {
            HSetState( (Handle) entityRefs,  sRefs );       assert(MemError() ==
noErr);
        }

        if (entityInfos != nil) {
            HSetState( (Handle) entityInfos, sInfos );      assert(MemError() ==
noErr);
```

```
        }
    }

    return err;
}
```

The next routine, shown in Listing 2-6, opens the database for reading, gets the entity references for all of the TCP/IP network connection entities in the default area (using the `MyGetEntitiesList` routine in Listing 2-5), and prints their user-visible names. This routine calls a routine, `MyGetEntityUserVisibleName`, which hasn't been documented yet. It is shown in Listing 2-9 in the section "Reading and Writing Preferences" (page 34).

Listing 2-6 Printing the user-visible name for an entity

```
static void PrintAllTCPEntityNames(void)
{
    OSStatus       err;
    CfgDatabaseRef dbRef;
    CfgAreaID      readArea;
    CfgEntityRef **entityRefs;
    ItemCount      entityCount;
    ItemCount      entityIndex;
    Str255         userVisibleName;

    entityRefs = (CfgEntityRef **) NewHandle(0);
    err = MemError();
    if (err == noErr) {
        err = MyOpenDatabaseForReading(&dbRef, &readArea);
        if (err == noErr) {
            err = MyGetEntitiesList(dbRef, readArea,
                    kOTCfgClassNetworkConnection, kOTCfgTypeTCPv4,
                    entityRefs, nil);
        }
        if (err == noErr) {
            HLock( (Handle) entityRefs );                      assert(MemError() ==
noErr);
            printf("List of TCP/IP Network Connection Entities\n");
            entityCount = GetHandleSize( (Handle) entityRefs ) / sizeof(CfgEntityRef);
            for (entityIndex = 0; entityIndex < entityCount; entityIndex++) {
                err = MyGetEntityUserVisibleName(dbRef,
                                            &(*entityRefs)[entityIndex],
                                            userVisibleName);
                if (err == noErr) {
                    printf("%ld) "%#s"\n", entityIndex, userVisibleName);
                }
            }
        }
        MyCloseDatabaseAfterReading(dbRef, readArea);
    }
    if (entityRefs != nil) {
        DisposeHandle( (Handle) entityRefs );                   assert(MemError() ==
noErr);
    }

    if (err != noErr) {
        printf("Failed with error %ld.\n", err);
    }
}
```

Finding an Active Entity

Currently, only one entity can be active for any given network connection type. This is not a restriction of Network Setup itself, but a limitation in the network protocol stacks. When you look for an active entity for a particular network protocol, you should be aware that, in the future, there may be more than one.

Because of the complexity of this algorithm and because its implementation relies on concepts that haven't been discussed yet, the steps and sample code for finding an active entity are shown in "Working with Sets" (page 42), later in this chapter.

Reading and Writing Preferences

Once you have an entity reference, reading and writing preferences in the entity is a straightforward exercise. The basic steps are to open the entity, read and write the desired preferences, and close the entity. This section describes this process for reading variable-length and fixed-size preferences and for writing preferences.

Reading Fixed-size Preferences

Many Network Setup preferences are of a fixed size. Reading a fixed size preference is easy because you simply read it into the C structure that corresponds to the preference. The code in Listing 2-7 shows a simple wrapper routine you can use to read a fixed size preference from an entity within the database. The `prefType` parameter controls the preference that is read. The preference data is put in the buffer described by `buffer` and `bufferSize`.

Listing 2-7 Reading a fixed-size preference

```
static OSStatus MyReadFixedSizePref(CfgDatabaseRef dbRef,
                                const CfgEntityRef *entity,
                                OSType prefType,
                                void *buffer,
                                ByteCount bufferSize)
{
    OSStatus err;
    OSStatus err2;
    CfgEntityAccessID accessID;

    assert(dbRef  != nil);
    assert(entity != nil);
    assert(buffer != nil);

    err = OTCfgOpenPrefs(dbRef, entity, false, &accessID);
    if (err == noErr) {
        err = OTCfgGetPrefs(accessID, prefType, buffer, bufferSize);
        err2 = OTCfgClosePrefs(accessID);
        if (err == noErr) {
            err = err2;
        }
    }
    return err;
}
```

Note
The sample shown in Listing 2-7, which opens and closes the entity before reading each preference, is implemented in an inefficient manner for the sake of clarity. If you are reading multiple preferences, it is more efficient to open the entity once. Then read the preferences by calling `OTCfgGetPrefs` or `OTCfgSetPrefs` multiple times and close the entity when you're done. ◆

A noteworthy point about reading preferences is that the `OTCfgOpenPrefs` function does not take an area parameter. This is because the `CfgEntityRef` itself implicitly includes the area. The significant of this point is demonstrated in the section "Working with Sets" (page 42).

You can use the `MyReadFixedSizePref` routine shown in Listing 2-7 to read specific preferences within an entity. For example, Listing 2-8 shows how to read the DHCP lease information from a TCP/IP network connection entity. The routine calls `MyReadFixedSizePref`, passing it the appropriate preference type (`kOTCfgTCPDHCPLeaseInfoPref`), a pointer to the corresponding C structure, and the size of the structure.

Listing 2-8 Reading the DHCP lease info preference in a TCP/IP network connection entity

```
static OSStatus MyReadDHCPLeaseInfo(CfgDatabaseRef dbRef,
                                const CfgEntityRef *entity,
                                OTCfgTCPDHCPLeaseInfo *dhcpInfo)
{
    OSStatus err;

    assert(dbRef    != nil);
    assert(entity   != nil);
    assert(dhcpInfo != nil);

    err = MyReadFixedSizePref(dbRef, entity, kOTCfgTCPDHCPLeaseInfoPref,
                            dhcpInfo, sizeof(*dhcpInfo));

    return err;
}
```

IMPORTANT

You can derive the C structure for a specific preference type by removing the "k" from the front of the name and the "Pref" from the end. For example, the C structure for `kOTCfgTCPDHCPLeaseInfoPref` is `OTCfgTCPDHCPLeaseInfo`. The preference type constants and preference structures for all of the Apple-defined preferences are provided in Chapter 4, "Network Setup Protocol Structures and Data Types." ▲

Reading Variable-size Preferences

The `MyReadFixedSizePref` routine shown in Listing 2-7 also works with variable size preferences that have a known maximum size that internally includes the size of the preference. The user-visible name preference (`kOTCfgUserVisibleNamePref`), which contains a packed Pascal string, is an example. The maximum length of a Pascal string is 256 bytes, and the first byte denotes the length of the actual string data. Listing 2-9 shows how to use `MyReadFixedSizePref` to read this type of variable size preference.

Listing 2-9 Reading the user-visible name preference

```
static OSStatus MyGetEntityUserVisibleName(CfgDatabaseRef dbRef,
                                    const CfgEntityRef *entity,
                                    Str255 name)
{
    OSStatus err;

    assert(dbRef != nil);
    assert(entity != nil);
    assert(name != nil);

    err = MyReadFixedSizePref(dbRef, entity, kOTCfgUserVisibleNamePref,
                            name, sizeof(Str255));
    return err;
}
```

If the variable size preference you want to read does not have a known maximum size and does not store its size internally, you need to know how big a buffer to allocate before you call `OTCfgGetPrefs`. You can get this information by calling `OTCfgGetPrefsSize` before you read the preference, as shown in Listing 2-10.

Listing 2-10 Calling OTCfgGetPrefsSize to read a variable-size preference

```
static OSStatus MyReadVariableSizePref(CfgDatabaseRef dbRef,
                                    const CfgEntityRef *entity,
                                    OSType prefType,
                                    Handle buffer)
{
    OSStatus err;
    OSStatus err2;
    CfgEntityAccessID accessID;
    ByteCount prefSize;
    SInt8 s;

    assert(dbRef  != nil);
    assert(entity != nil);
    assert(buffer != nil);

    err = OTCfgOpenPrefs(dbRef, entity, false, &accessID);
    if (err == noErr) {
        err = OTCfgGetPrefsSize(accessID, prefType, &prefSize);
        if (err == noErr) {
            SetHandleSize(buffer, prefSize);
            err = MemError();
        }
        if (err == noErr) {
            s = HGetState(buffer);                      assert(MemError()
== noErr);
            HLock(buffer);                              assert(MemError()
== noErr);
            err = OTCfgGetPrefs(accessID, prefType, *buffer, prefSize);
            HSetState(buffer, s);                       assert(MemError()
== noErr);
        }
        err2 = OTCfgClosePrefs(accessID);
        if (err == noErr) {
            err = err2;
        }
    }
    return err;
}
```

Writing Preferences

Listing 2-11 shows the routine `MyWritePref`, which demonstrates the basic mechanism for writing preferences. Writing a preference is similar to reading a preference, with the following exceptions:

- When you open the entity, open the entity for writing by passing `true` in the `writer` parameter of `OTCfgOpenPrefs`.
- The entity that is opened must be in a writable temporary area. Attempting to open for writing an entity in a read-only area will result in an error.

Note
You don't need provide the area identifier when you call `OTCfgOpenPrefs` because an entity "knows" the area to which it belongs. ◆

Listing 2-11 Writing a preference

```
static OSStatus MyWritePref(CfgDatabaseRef dbRef,
                        const CfgEntityRef *entity,
                        OSType prefType,
                        const void *buffer,
                        ByteCount bufferSize)
{
    OSStatus err;
    OSStatus err2;
    CfgEntityAccessID accessID;

    assert(dbRef      != nil);
    assert(entity     != nil);
    assert(buffer     != nil);

    err = OTCfgOpenPrefs(dbRef, entity, true, &accessID);
    if (err == noErr) {
        err = OTCfgSetPrefs(accessID, prefType, buffer, bufferSize);
        err2 = OTCfgClosePrefs(accessID);
        if (err == noErr) {
            err = err2;
        }
    }
    return err;
}
```

Note
The sample shown in Listing 2-11, which opens and closes the entity for each preference written, is implemented in an inefficient manner for the sake of clarity. If you are writing multiple preferences, it is more efficient to open the entity, write your preferences by calling `OTCfgSetPrefs` multiple times, and close the entity when you're done. ◆

Iterating the Preferences in an Entity

Network Setup provides functions for iterating all of the preferences in an entity. You will rarely need to do this, but the code in Listing 2-12 gives an example. The code first calls `OTCfgGetPrefsTOCCount` (TOC stands for "Table of Contents") to get a count of the number of preferences in the entity and then calls `OTCfgGetPrefsTOC` to get an array of `CfgPrefsHeader` structures. Each structure represents a preference in the entity, with fields for the preference's type and size.

Listing 2-12 Printing an entity's table of contents

```
static void PrintPrefsTOC(CfgDatabaseRef dbRef, const CfgEntityRef *entity)
{
    OSStatus err;
    OSStatus err2;
    CfgEntityAccessID accessID;
    ItemCount prefsTOCCount;
    ItemCount prefsTOCIndex;
    CfgPrefsHeader *prefsTOC;
    OSType    prefType;
    ByteCount prefSize;
    assert(dbRef  != nil);
    assert(entity != nil);

    prefsTOC = nil;

    err = OTCfgOpenPrefs(dbRef, entity, false, &accessID);
    if (err == noErr) {
        err = OTCfgGetPrefsTOCCount(accessID, &prefsTOCCount);
        if (err == noErr) {
            prefsTOC = (CfgPrefsHeader *) NewPtr(prefsTOCCount * 
sizeof(CfgPrefsHeader));
            err = MemError();
        }
        if (err == noErr) {
            err = OTCfgGetPrefsTOC(accessID, &prefsTOCCount, prefsTOC);
        }
        if (err == noErr) {
            for (prefsTOCIndex = 0; prefsTOCIndex < prefsTOCCount; prefsTOCIndex++) {
                prefType = prefsTOC[prefsTOCIndex].fType;
                prefSize = prefsTOC[prefsTOCIndex].fSize;
                printf("type = '%4.4s', size = %ld\n", &prefType, prefSize);
            }
        }

        err2 = OTCfgClosePrefs(accessID);
        if (err == noErr) {
            err = err2;
        }
    }

    if (prefsTOC != nil) {
        DisposePtr( (Ptr) prefsTOC );                           assert(MemError() == 
noErr);
    }

    if (err != noErr) {
        printf("Failed with error %ld.\n", err);
    }
}
```

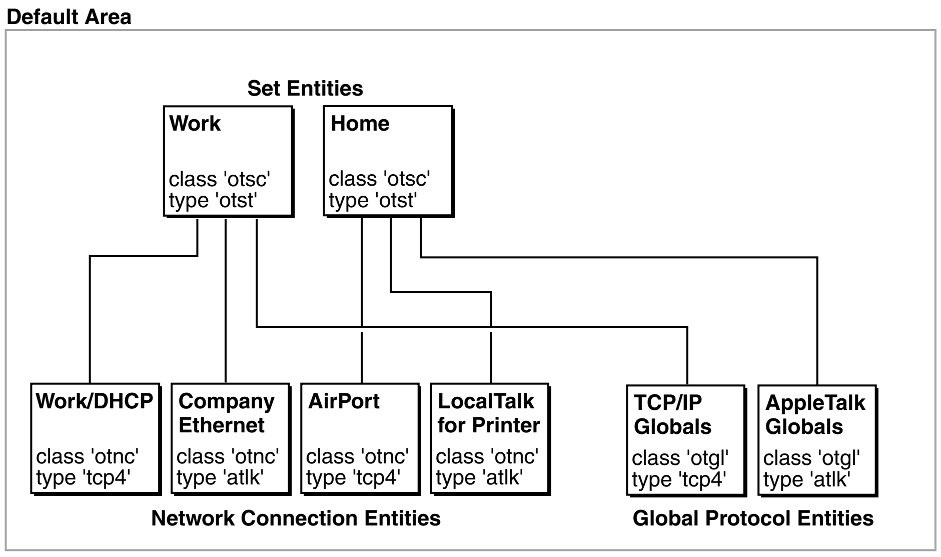
Working with Sets

The Network Setup database uses set entities to store collections of other entity references. When network entities are grouped into sets, they can be activated and deactivated as a group. All of the network entities in all of the sets reside in a single area, so there are no limits on the way entities can be grouped. For example, a single network connection entity can be referenced by multiple sets.

IMPORTANT
Sets contain entity references — not the entities themselves. ▲

Figure 2-1 shows the relationship between set entities, network connection entities, and global protocol entities.

Figure 2-1 Set entities reference other entities



There are a few basic rules for set entities:

- Each set entity contains a preference, `kOTCfgSetsStructPref`, that has a flag that determines whether the set is active.
- At all times, there must be one and only one active set.
- Each set entity contains a preference, `kOTCfgSetsVectorPref`, that includes, as elements of an unbounded array, the entity references of all entities in the set.
- For legacy synchronization to work correctly, each set entity must contain one and only one entity of each type of network connection and global protocol entity. This restriction will be relaxed in future versions of Mac OS.

When you make changes to a set entity, you must follow these rules:

- If you mark a set as active, you must deactivate the previously active set.
- When you add an entity to a set entity, you must remove the first entity in the array of the same class and type as the entity you are adding. If there is more than one entity of the same class and type, you can safely leave the other entities in the set entity because you are running on a system that supports multihoming.
- When you delete an entity, you must delete its reference from all set entities, whether they are active or not.
- You must not delete the last remaining entity of a particular class and type from a set entity.
- Do not add a set entity reference to another set entity. Network Setup does not support nested set entities.

Finding the Active Set Entity

The basic algorithm for finding the active entity of a particular class and type is:

1. Get a list of all set entities.
2. Search the list for the active set entity.
3. Get the contents of that set entity. (The contents of a set entity is a list of entity references.)
4. Search the set's entity references for the entity reference having the appropriate class and type.

The MyFindActiveSet routine in Listing 2-13 implements the first two steps. It starts by getting a list of all of the set entities by calling the MyGetEntitiesList routine (Listing 2-5). Then MyFindActiveSet iterates through all of the set entities, reading the kOTCfgSetsStructPref preference of each set entity. That preference maps to the CfgSetsStruct structure, which contains an fFlags member. One bit of the fFlags member, kOTCfgSetsFlagActiveMask, indicates whether this set entity is the active set entity. If it is, the routine breaks out of the loop and returns the set's entity reference to the caller.

Listing 2-13 Finding the active set entity

```
static OSStatus MyFindActiveSet(CfgDatabaseRef dbRef,
                            CfgAreaID area,
                            CfgEntityRef *activeSet)
{
    OSStatus        err;
    CfgEntityRef  **entityRefs;
    ItemCount       entityCount;
    ItemCount       entityIndex;
    Boolean         found;
    CfgSetsStruct   thisStruct;

    assert(dbRef != nil);
    assert(area  != kInvalidCfgAreaID);

    entityRefs = (CfgEntityRef  **) NewHandle(0);
    err = MemError();
    if (err == noErr) {
        err = MyGetEntitiesList(dbRef, area,
                            kOTCfgClassSetOfSettings, kOTCfgTypeSetOfSettings,
                            entityRefs, nil);
    }

    if (err == noErr) {
        HLock( (Handle) entityRefs );                           assert(MemError() ==
noErr);
        entityCount = GetHandleSize( (Handle) entityRefs ) / sizeof(CfgEntityRef);
        found = false;
        for (entityIndex = 0; entityIndex < entityCount; entityIndex++) {
            err = MyReadFixedSizePref(dbRef, &(*entityRefs)[entityIndex],
                                kOTCfgSetsStructPref,
                                &thisStruct, sizeof(thisStruct));
            if ((err == noErr) && ((thisStruct.fFlags & kOTCfgSetsFlagActiveMask) != 
0)) {
                found = true;
                break;
            }
            if (err != noErr) {
                break;
            }
        }
        if ( ! found ) {
            err = kCfgErrEntityNotFound;
        }
    }
    if (err == noErr) {
        *activeSet = (*entityRefs)[entityIndex];
    }

    if (entityRefs != nil) {
        DisposeHandle( (Handle) entityRefs );                   assert(MemError() == 
noErr);
    }

    return err;
}
```

The remaining two steps for finding the set entity of a particular class and type are implemented by the `MyFindFirstActiveEntity` routine, shown in Listing 2-14. It calls `MyFindActiveSet` (Listing 2-13) to find the entity reference of the active set entity. The `MyFindFirstActiveEntity` routine then reads the `kOTCfgSetsVectorPref` preference out of the active set entity. This preference is a count field followed by an unbounded array of `CfgSetsElement` structures, each of which represents an entity in the set. Because of its variable size, `MyFindFirstActiveEntity` reads the preference by calling `MyReadVariableSizePref` (Listing 2-10). Once it has the array of information about entities contained in the set, `MyFindFirstActiveEntity` iterates over that array looking for the first element whose class and type matches the required class and type specified by the caller. When it finds the correct entity in the set, `MyFindFirstActiveEntity` breaks out of the loop and returns the found entity reference to the caller.

Listing 2-14 Finding the active entity of a given class and type

```
static OSStatus MyFindFirstActiveEntity(CfgDatabaseRef dbRef,
                                    CfgAreaID area,
                                    OSType entityClass,
                                    OSType entityType,
                                    CfgEntityRef *activeEntity)
{
    OSStatus      err;
    CfgEntityRef  activeSet;
    CfgSetsVector **entitiesInSet;
    ItemCount     entityIndex;
    Boolean       found;
    CfgEntityInfo thisEntityInfo;

    entitiesInSet = (CfgSetsVector **) NewHandle(0);
    err = MemError();
    if (err == noErr) {
        err = MyFindActiveSet(dbRef, area, &activeSet);
    }
    if (err == noErr) {
        err = MyReadVariableSizePref(dbRef, &activeSet, kOTCfgSetsVectorPref, (Handle )
                                entitiesInSet);
    }
    if (err == noErr) {
        HLock( (Handle) entitiesInSet );

        found = false;
        for (entityIndex = 0; entityIndex < (**entitiesInSet).fCount; entityIndex++) {
            thisEntityInfo = (**entitiesInSet).fElements[entityIndex].fEntityInfo;
            found = ( thisEntityInfo.fClass == entityClass && thisEntityInfo.fType ==
entityType );
            if (found) {
                break;
            }
        }
        if ( ! found ) {
            err = kCfgErrEntityNotFound;
        }
    }
    if (err == noErr) {
        *activeEntity = (**entitiesInSet).fElements[entityIndex].fEntityRef;
        OTCfgChangeEntityArea(activeEntity, area);
    }

    if (entitiesInSet != nil) {
        DisposeHandle( (Handle) entitiesInSet );                assert(MemError() == 
noErr);
    }
    return err;
}
```

The code in Listing 2-15 pulls together the process of finding an active set entity by finding the active TCP/IP set entity. It opens the database, calls `MyFindFirstActiveEntity` (Listing 2-14) with `kOTCfgClassNetworkConnection` and `kOTCfgTypeTCPv4` as parameters, calls `MyGetEntityUserVisibleName` (Listing 2-9) to get and print the entity's user visible name, and calls `MyCloseDatabaseAfterReading` (Listing 2-3) to close the database.

Listing 2-15 Finding the active TCP/IP entity

```
static void PrintActiveTCPEntity(void)
{
    OSStatus       err;
    CfgDatabaseRef dbRef;
    CfgAreaID      readArea;
    CfgEntityRef   activeTCPEntity;
    Str255         userVisibleName;

    err = MyOpenDatabaseForReading(&dbRef, &readArea);
    if (err == noErr) {
        err = MyFindFirstActiveEntity(dbRef, readArea,
                                  kOTCfgClassNetworkConnection,
                                  kOTCfgTypeTCPv4,
                                  &activeTCPEntity);
        if (err == noErr) {
            err = MyGetEntityUserVisibleName(dbRef, &activeTCPEntity, userVisibleName);
        }
        if (err == noErr) {
            printf("User-visible name of active TCP/IP entity = "%#s"\n",
userVisibleName);
        }

        MyCloseDatabaseAfterReading(dbRef, readArea);
    }

    if (err != noErr) {
        printf("Failed with error %ld.\n", err);
    }
}
```

Areas and Sets

When working with sets you need to be very careful about area identifiers. There are three key points to remember:

- The area identifier is embedded in the entity reference.
- All modifications to the database are done in a temporary area.
- The temporary area is destroyed when changes are committed to the database.

So, the area identifier that is embedded in the entity reference in a set is an area identifier for an area that no longer exists. This can cause your software to fail. For example, consider the following sequence:

1. Start with a database whose default area identifier is 1370.
2. A program opens the database for writing, which creates a temporary area whose identifier is 6288.
3. The program adds an entity reference to the active set entity in the temporary area. Because the added entity reference describes an entity in the temporary area, its area identifier is 6288.
4. The writing program commits its changes to the database, overwriting area 1370 with the content of area 6288. The active set entity in area 1370 now contains an entity reference whose area identifier is 6288.
5. Your program opens the database for reading. It then opens the active set entity and reads the entity references contained therein. When it tries to use one of those entities, the program fails because the entity's area identifier is 6288, not 1370.

The solution to this problem is very simple: assume that all entity references in a set refer to entities that are in the same area as the set. This has two practical consequences.

- When comparing two entity references that might have come from a set entity, always pass `kOTCfgIgnoreArea` when calling `OTCfgIsSameEntityRef`. The `OTCfgIsSameEntityRef` function will then compare the entities as if they were in the same area.
- When opening an entity whose reference you have obtained from a set, always call `OTCfgChangeEntityArea` to reset its area identifier to that of the area in which you are working.

Listing 2-14 demonstrates this technique.

Protocol-specific Topics

This section contains hints and tips for working with the Network Setup preferences of certain protocol stacks provided by Apple.

TCP/IP Notes

A TCP/IP network connection entity has a class of `kOTCfgClassNetworkConnection` and a type of `kOTCfgTypeTCPv4`. The entity must contain the following preferences:

- `kOTCfgTCPInterfacesPref`, which contains the core TCP/IP configuration information. For details, see the discussion below and `OTCfgTCPInterfacesUnpacked` (page 112), `OTCfgTCPInterfacesPacked` (page 114), and `OTCfgTCPInterfacesPackedPart` (page 114).
- `kOTCfgTCPDeviceTypePref`, which contains data needed by the current TCP/IP control panel. For details, see `OTCfgTCPDeviceTypePref` in the section “TCP/IP Constants and Other Data Types” (page 160).
- `kOTCfgTCPRoutersListPref`, which contains the list of configured routers. For details, see `OTCfgTCPRoutersList` (page 119).
- `kOTCfgTCPSearchListPref`, which contains the strings which make up the implicit search path for DNS. For details, see `OTCfgTCPSearchList` (page 120).

- kOTCfgTCPDNSServersListPref, which contains the list of configured DNS servers. For details, see OTCfgTCPDNSServersList (page 116).
- kOTCfgTCPSearchDomainsPref, which contains the list of additional domains to be searched. For details, see OTCfgTCPSearchDomains (page 120).
- kOTCfgTCPUnloadAttrPref, which specifies how TCP/IP loads and unloads. For details, see OTCfgTCPUnloadAttr (page 121).
- kOTCfgTCPLocksPref, which is used by the TCP/IP control panel to remember which preferences are locked. For details, see OTCfgTCPLocks (page 116).

The only complex preference in a TCP/IP network connection entity is the kOTCfgTCPInterfacesPref preference. The data for this preference is packed in an unusual way that makes the preference tricky to access from C. To help solve this problem, Network Setup declares two sets of C structures for this preference.

- OTCfgTCPInterfacesPacked and OTCfgTCPInterfacesPackedPart help you access the preference in its packed format.
- OTCfgTCPInterfacesUnpacked is an unpacked form of the preference that you can use internally within your code. When you read the preference, you can unpack it into this structure. You can then manipulate the unpacked structure and only pack it again when you write it.

Listing 2-16 provides sample code that unpacks and packs a kOTCfgTCPInterfacesPref preference.

Listing 2-16 Packing and unpacking the kOTCfgTCPInterfacesPref preference

```
static OSStatus MyPackTCPInterfacesPref(const OTCfgTCPInterfacesUnpacked *unpackedPref,
                                        OTCfgTCPInterfacesPacked *packedPref,
                                        ByteCount *packedPrefSize)
{
    UInt8 *cursor;

    assert(unpackedPref   != nil);
    assert(packedPref     != nil);
    assert(packedPrefSize != nil);

    // Start the cursor at the beginning of the packed preference.
    cursor = (UInt8 *) packedPref;

    // For each field in the unpacked pref, copy the field to the
    // packed preference cursor and advance the cursor appropriately.
    if (unpackedPref->fCount != 1) goto prefDataErr;
    *((UInt16 *) cursor) = unpackedPref->fCount;
    cursor += sizeof(UInt16);

    *cursor = unpackedPref->fConfigMethod;
    cursor += sizeof(UInt8);

    *((InetHost *) cursor) = unpackedPref->fIPAddress;
    cursor += sizeof(InetHost);
    *((InetHost *) cursor) = unpackedPref->fSubnetMask;
    cursor += sizeof(InetHost);

    // Writing an AppleTalk zone longer than 32 characters is an error.

    if ( unpackedPref->fAppleTalkZone[0] > 32 ) goto prefDataErr;

    BlockMoveData(unpackedPref->fAppleTalkZone, cursor, unpackedPref->fAppleTalkZone[0]
+ 1);
    cursor += (unpackedPref->fAppleTalkZone[0] + 1);
    BlockMoveData(unpackedPref->path, cursor, 36);
    cursor += 36;
    BlockMoveData(unpackedPref->module, cursor, 32);
    cursor += 32;
    *((UInt32 *) cursor) = unpackedPref->framing;
    cursor += sizeof(UInt32);

    // Now calculate the packed preference size by taking the difference
    // between the final cursor position and the initial cursor position.
    *packedPrefSize = (cursor - ((UInt8 *) packedPref));

    return noErr;

prefDataErr:
    return paramErr;
}
```

```
static OSStatus MyUnpackTCPInterfacesPref(const OTCfgTCPInterfacesPacked *packedPref,
                                    ByteCount packedPrefSize,
                                    OTCfgTCPInterfacesUnpacked *unpackedPref)
{
    UInt8 *cursor;

    assert(packedPref   != nil);
    assert(unpackedPref != nil);

    // Put the cursor at the beginning of the packed preference data.
    cursor = (UInt8 *) packedPref;

    // Walk through the packed preference data and extract the fields.
    unpackedPref->fCount = *((UInt16 *) cursor);

    if (unpackedPref->fCount != 1) goto prefDataErr;
    cursor += sizeof(UInt16);

    unpackedPref->fConfigMethod = *cursor;
    cursor += sizeof(UInt8);

    // The following code accesses a long off a word.
    // Network Setup is PowerPC only, and the emulated PowerPC processor handles these
    // misaligned accesses.
    unpackedPref->fIPAddress = *((InetHost *) cursor);
    cursor += sizeof(InetHost);
    unpackedPref->fSubnetMask = *((InetHost *) cursor);
    cursor += sizeof(InetHost);

    // fAppleTalkZone is a Str32.  A longer string in the 'iitf' preference causes an
error.
    if ( *cursor > 32 ) goto prefDataErr;

    BlockMoveData(cursor, unpackedPref->fAppleTalkZone, *cursor + 1);
    cursor += (*cursor + 1);
    BlockMoveData(cursor, unpackedPref->path, 36);
    cursor += 36;
    BlockMoveData(cursor, unpackedPref->module, 32);
    cursor += 32;
    unpackedPref->framing = *((UInt32 *) cursor);
    cursor += sizeof(UInt32);
    // If the cursor doesn’t stop at the end of the packed preference data, a data
format error occurs.
    if ( (cursor - ((UInt8 *) packedPref)) != packedPrefSize) goto prefDataErr;

    return noErr;

prefDataErr:
    return paramErr;
}
```

Remote Access Notes

A Remote Access network connection entity has a class of `kOTCfgClassNetworkConnection` and a type of `kOTCfgTypeRemote` and typically contains the following preferences:

- `kOTCfgRemoteConnectPref`, which contains core connection preferences. For details, see `OTCfgRemoteConnect` (page 127).
- `kOTCfgRemoteUserPref`, which contains the user name. For details, see `OTCfgRemoteUserPref` in the section “Apple Remote Access Constants and Other Data Types” (page 163).
- `kOTCfgRemotePasswordPref`, which contains the user’s encrypted password. For details, see the sample code below and `OTCfgRemotePassword` (page 135).
- `kOTCfgRemoteAddressPref`, which contains the phone number to dial. For details, see `OTCfgRemoteAddressPref` in the section “Apple Remote Access Constants and Other Data Types” (page 163).
- `kOTCfgRemoteDialingPref`, which contains redial preferences. For details, see `OTCfgRemoteDialing` (page 131).
- `kOTCfgRemoteClientMiscPref`, which controls the “dial on demand” feature of IPCP. For details, see `OTCfgRemoteClientMisc` (page 127).
- `kOTCfgRemoteIPCPPref`, which contains low-level preferences for IPCP. You typically set this preference to a default value obtained by calling `OTCfgGetDefault` (page 90). For details, see `OTCfgRemoteIPCP` (page 132).
- `kOTCfgRemoteLCPPref`, which contains low-level preferences for LCP. You typically set this preference to a default value obtained by calling `OTCfgGetDefault` (page 90). For details, see `OTCfgRemoteLCP` (page 133).

- `kOTCfgRemoteLogOptionsPref`, which contains the "verbose logging" option. For details, see `OTCfgRemoteLogOptions` (page 135).
- `kOTCfgRemoteClientLocksPref`, which is used by the Remote Access control panel to remember which preferences are locked. For details, see `OTCfgRemoteClientLocks` (page 125).

To create the `kOTCfgRemotePasswordPref`, you must encrypt the user's password. The code in Listing 2-17 shows a technique for doing this.

Listing 2-17 Encrypting the user's password

```
static void EncodeRemotePasswordNetworkSetup(
                            ConstStr255Param userName,
                            ConstStr255Param password,
                            Str255 encodedPassword)
{
    BlockZero(encodedPassword, sizeof(Str255));
    BlockMoveData(password + 1, encodedPassword, password[0]);

    (void) OTCfgEncrypt( (UInt8 *) userName,
                        encodedPassword,
                        sizeof(Str255));
}
```

Modem Notes

A Remote Access network connection entity has a class of `kOTCfgClassNetworkConnection` and a type of `kOTCfgTypeModem`. The entity typically contains the following preferences:

- `kOTCfgModemGeneralPrefs`, which contains the core modem preferences. For details, see `OTCfgModemGeneral` (page 141).
- `kOTCfgModemLocksPref`, which is used by the Modem control panel to remember which preferences are locked. For details, see `OTCfgModemLocks` (page 143).

When creating the `kOTCfgModemGeneralPrefs` preference, you have to supply the name of a serial port that is visible to Open Transport. For information about building a list of Open Transport serial ports and their user-visible names, see

DTS Technote 1119 Serial Port Apocrypha available at http://developer.apple.com/technotes/tn/tn1119.html.

Notes for Third Parties

This section contains miscellaneous hints and tips for third-party developers who want to use Network Setup to store their own preferences.

Storing Third-party Preferences in Apple Entities

It is reasonable for third-party developers to store custom preferences inside Apple Computer's protocol entities. For example, a TCP/IP virtual private network (VPN) implementation might store per-connection preferences inside Apple Computer's TCP/IP network connection entity. This is perfectly legal — in fact it is encouraged — but you need to follow one important rule: The preference type for your preference must be registered as a unique creator code with DTS at `http://developer.apple.com/dev/cftype`. Registering preference types will prevent two different developers from using the same preference type for conflicting preferences.

Network Setup and Third-party Protocol Stacks

If you're writing a third-party protocol stack, you can use the Network Setup database to store your preferences in much the same way as the Apple protocol stacks do. There are a few important things to remember.

- It is recommended that you use the existing classes, `kOTCfgClassNetworkConnection` and `kOTCfgClassGlobalSettings`, for your global protocol and network connection entities.
- You should register a unique creator code with DTS at `http://developer.apple.com/dev/cftype` and use it as the type for your entities. This will ensure that your work does not conflict with Apple Computer or other developers.

Your protocol stack configurator should call `OTCfgInstallNotifier` (page 94) to install a Network Setup notifier to watch for changes to its preferences by your control panel or by third-party software.

Network Setup Reference

This chapter describes the functions, structures, and data types for calling Network Setup. For protocol-specific preferences, see Chapter 4, "Network Setup Protocol Structures and Data Types."

Network Setup Functions

The Network Setup functions are described in these sections:

- "Opening and Closing the Network Setup Database" (page 57)
- "Managing Areas" (page 59)
- "Managing Entities" (page 72)
- "Managing Preferences" (page 83)
- "Preference Utilities" (page 92)
- "Installing and Removing a Notification Callback" (page 94)

Opening and Closing the Network Setup Database

Before attempting to call the Network Setup functions, you must open the Network Setup database. Be sure to close the database when you are done. The functions are:

- `OTCfgOpenDatabase` (page 58) opens the Network Setup database.
- `OTCfgCloseDatabase` (page 58) closes the Network Setup database.

OTCfgOpenDatabase

Opens a session with the Network Setup database.

```
OSStatus OTCfgOpenDatabase (CfgDatabaseRef* dbRef);
```

`dbRef`	On input, a pointer to a value of type `CfgDatabaseRef` (page 98). On output, `dbRef` is a reference to the opened database that is passed as a parameter to other Network Setup functions.
function result	A value of `noErr` if the database was opened. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgOpenDatabase` function opens a session with the Network Setup database. Your application must call `OTCfgOpenDatabase` successfully before it can call any other Network Setup function.

OTCfgCloseDatabase

Closes the Network Setup database.

```
OSStatus OTCfgCloseDatabase (OTCfgDatabaseRef* dbRef);
```

`dbRef`	A pointer to value of type `CfgDatabaseRef` (page 98) that represents the database session you want to close.
function result	A value of `noErr` if the database was closed. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgCloseDatabase` function closes the database session represented by `dbRef`.

Note
Closing a database session automatically removes any notification callback that has been installed for the session represented by `dbRef`. ◆

Managing Areas

The following functions manage areas in the Network Setup database:

- `OTCfgGetCurrentArea` (page 60) obtains the default area in the database.
- `OTCfgSetCurrentArea` (page 61) sets the default area in the database.
- `OTCfgOpenArea` (page 61) opens an area in the database.
- `OTCfgCloseArea` (page 62) closes an area in the database.
- `OTCfgBeginAreaModifications` (page 63) creates a temporary area for modifying the database.
- `OTCfgCommitAreaModifications` (page 64) commits changes made in a temporary area to the database.
- `OTCfgAbortAreaModifications` (page 65) discards a temporary area and all modifications made to it.
- `OTCfgIsSameAreaID` (page 65) determines whether two area IDs are the same.
- `OTCfgGetAreaName` (page 66) gets the name of an area in the database.
- `OTCfgSetAreaName` (page 67) sets the name of an area in the database.
- `OTCfgGetAreasCount` (page 68) obtains the number of areas in the database.
- `OTCfgGetAreasList` (page 68) obtains the area IDs and area names in the database.
- `OTCfgCreateArea` (page 70) creates a new area in the database.
- `OTCfgDuplicateArea` (page 71) copies the contents of an area to another area.
- `OTCfgDeleteArea` (page 72) deletes an area in the database.

IMPORTANT

Areas other than the default area (also known as the current area) do not affect any network setting, so many of the area manipulation functions described in this section are not commonly used. You rarely need to call `OTCfgGetAreaName`, `OTCfgSetAreaName`, `OTCfgGetAreasCount`, `OTCfgGetAreasList`, `OTCfgCreateArea`, `OTCfgDuplicateArea`, or `OTCfgDeleteArea`. ▲

OTCfgGetCurrentArea

Obtains the default area.

```
OSStatus OTCfgGetCurrentArea (CfgDatabaseRef dbRef,
                              CfgAreaID* areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a pointer to a value of type `CfgAreaID` (page 98). On output, `areaID` points to the area ID of the current area.
function result	A value of `noErr` indicates that `OTCfgGetCurrentArea` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgGetCurrentArea` function obtains the area ID of the default area. The default area is sometimes referred to as the current area.

OTCfgSetCurrentArea

Sets the default area.

```
OSStatus OTCfgSetCurrentArea (CfgDatabaseRef dbRef,
                              CfgAreaID areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) containing the `areaID` that identifies the area that is to be made active. If the area does not exist, `OTCfgSetCurrentArea` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgSetCurrentArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgSetCurrentArea` function makes the area ID specified by the `areaID` parameter the default area. The default area is sometimes referred to as the current area.

▲ **WARNING**
Do not change the default area. If you want to modify settings, make changes to the entities within the default area. ▲

OTCfgOpenArea

Opens an area in the Network Setup database for reading.

```
OSStatus OTCfgOpenArea (CfgDatabaseRef dbRef,
                              CfgAreaID areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be opened. If the area specified by `areaID` does not exist, `OTCfgOpenArea` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgOpenArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgOpenArea` function opens the specified area in the Network Setup database for reading.

OTCfgCloseArea

Closes an area in the Network Setup database.

```
OSStatus OTCfgCloseArea (CfgDatabaseRef dbRef,
                         CfgAreaID areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be closed. If the area specified by `areaID` does not exist, `OTCfgCloseArea` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgCloseArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgCloseArea` function closes an area in the database that was previously opened for reading by calling `OTCfgOpenArea` (page 61).

OTCfgBeginAreaModifications

Creates a temporary area for modifying an area.

```
OSStatus OTCfgBeginAreaModifications (CfgDatabaseRef dbRef,
                         CfgAreaID readAreaID,
                         CfgAreaID* writeAreaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`readAreaID`	On input, a value of type `CfgAreaID` (page 98) obtained by calling `OTCfgGetCurrentArea` (page 60). If the area specified by `readAreaID` does not exist, `OTCfgBeginAreaModifications` returns `kCfgErrAreaNotFound`.
`writeAreaID`	On input, a pointer to a value of type `CfgAreaID` (page 98). On output, `writeAreaID` points to a new area ID that your application should use to modify, delete, enumerate, or read data in the area.
function result	A value of `noErr` indicates that `OTCfgBeginAreaModifications` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgBeginAreaModifications` function creates a temporary area and returns in the `writeAreaID` parameter an area ID for it. The area ID for the temporary area can be passed as a parameter to subsequent calls for creating or modifying entities in the temporary area.

If you need to read the area’s original, unmodified data, you can continue using `readAreaID` to do so.

IMPORTANT

Only one program can open an area of writing at any one time. If another program has already opened the area for writing, `OTCfgBeginAreaModifications` returns `kCfgErrConfigLocked`. ▲

Call `OTCfgCommitAreaModifications` (page 64) to write the temporary area to the area identified by `writeAreaID`, dispose of the temporary area, and close the area represented by `writeAreaID`, or call `OTCfgAbortAreaModifications` (page 65) to close the area represented by `readAreaID` and discard the temporary area.

OTCfgCommitAreaModifications

Closes an area for writing and commits modifications.

```
OSStatus OTCfgCommitAreaModifications (CfgDatabaseRef dbRef,
                        CfgAreaID readAreaID,
                        CfgAreaID writeAreaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`readAreaID`	On input, a value of type `CfgAreaID` (page 98). If `readAreaID` does not exist or does not match the `writeAreaID` referred to by `OTCfgBeginAreaModifications` (page 63), `OTCfgCommitAreaModifications` returns `kCfgErrAreaNotFound`.
`writeAreaID`	On input, a value of type `CfgAreaID` (page 98) previously obtained by calling `OTCfgBeginAreaModifications` (page 63). If `writeAreaID` does not exist or does not match the `readAreaID` passed to `OTCfgBeginAreaModifications` (page 63), `OTCfgCommitAreaModifications` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgCommitAreaModifications` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgCommitAreaModifications` function writes the temporary area represented by `writeAreaID` to the area represented by `readAreaID` and closes the write area.

Readers of the area represented by `readAreaID` are informed that the database has been modified.

OTCfgAbortAreaModifications

Closes an area for writing without committing modifications.

```
OSStatus OTCfgAbortAreaModifications (CfgDatabaseRef dbRef,
                         CfgAreaID readAreaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`readAreaID`	On input, a value of type `CfgAreaID` (page 98) that identifies an area that has been opened for writing. If `readAreaID` does not exist or you have not called `OTCfgBeginAreaModifications` (page 63) for the area represented by `readAreaID`, `OTCfgAbortAreaModifications` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgAbortAreaModifications` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgAbortAreaModifications` function closes an area that was opened for writing without writing the modifications to the area presented by `readAreaID`.

OTCfgIsSameAreaID

Compares two area IDs.

```
Boolean OTCfgIsSameAreaID (CfgAreaID areaID1,
                         CfgAreaID areaID2);
```

`areaID1`	On input, a value of type `CfgAreaID` (page 98) representing one of the area IDs that is to be compared.
`areaID2`	On input, a value of type `CfgAreaID` (page 98) representing the other area ID that is to be compared.

function result	A Boolean value that is `TRUE` if the area IDs are the same and `FALSE` if the area IDs are different.

DISCUSSION

The `OTCfgIsSameAreaID` function determines whether two area IDs represent to the same area.

OTCfgGetAreaName

Obtains the user-visible name of an area.

```
OSStatus OTCfgGetAreaName (CfgDatabaseRef dbRef,
                          CfgAreaID areaID,
                          Str255 areaName);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area whose name is to be obtained. If the area specified by `areaID` does not exist, `OTCfgGetAreaName` returns `kCfgErrAreaNotFound`.
`areaName`	On input, a value of type `Str255`. On output, `areaName` contains the user-visible name of the area specified by `areaID`.
function result	A value of `noErr` indicates that `OTCfgGetAreaName` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetAreaName` function gets the user-visible name of the specified area.

Note
The `OTCfgGetAreaName` function is available in Network Setup version 1.0.1 and later. ◆

OTCfgSetAreaName

Sets the user-visible name of an area.

```
OSStatus OTCfgSetAreaName (CfgDatabaseRef dbRef,
                           CfgAreaID areaID,
                           ConstStr255Param areaName,
                           CfgAreaID* newAreaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area whose name is to be set. If the area specified by `areaID` does not exist, `OTCfgSetAreaName` returns `kCfgErrAreaNotFound`.
`areaName`	On input, a value of type `ConstStr255Param` that specifies the name to set. If an area of the name specified by `areaName` already exists, `OTCfgSetAreaName` returns `kCfgErrAreaAlreadyExists`.
`newAreaID`	On input, a pointer to value of type `CfgAreaID` (page 98). On output, `newAreaID` points to a new area ID that your application should use for any subsequent calls for the area.
function result	A value of `noErr` indicates that `OTCfgSetAreaName` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgSetAreaName` function changes the user-visible name of the specified area and returns a new area ID for that area.

▲ **WARNING**

Do not change the name of the default area. ▲

OTCfgGetAreasCount

Obtains the number of areas in the Network Setup database.

```
OSStatus OTCfgGetAreasCount (CfgDatabaseRef dbRef,
                       ItemCount* itemCount);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`itemCount`	On input, a pointer to value of type `ItemCount`. On output, `itemCount` points to the number of areas in the database.
function result	A value of `noErr` indicates that `OTCfgGetAreasCount` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetAreasCount` function obtains the number of areas that are currently defined in the database. Having the number of areas allows you to call `OTCfgGetAreasList` (page 68) to get the ID and name of each area.

OTCfgGetAreasList

Obtains the IDs and names of areas in the Network Setup database.

```
OSStatus OTCfgGetAreasList (CfgDatabaseRef dbRef,
                       ItemCount* itemCount,
                       CfgAreaID areaID[],
                       Str255 areaName[]);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`itemCount`	On input, a pointer to a value of type `ItemCount` that specifies the number of areas for which information is requested. Call `OTCfgGetAreasCount` (page 68) to determine the number of areas

	that are available. On output, `itemCount` points to the number of areas for which information was actually returned, which may be less that expected if areas were deleted between calling `OTCfgGetAreasCount` (page 68) and calling `OTCfgGetAreasList`.
`areaID`	On input, an array of elements of type `CfgAreaID` (page 98) that is large enough to hold the number of area IDs specified by `itemCount`. On output, each array element contains an area ID. If you don't want to get area IDs, set `areaID` to `NULL`.
`areaName`	On input, an array of elements of type `Str255` that is large enough to hold the number of area names specified by `itemCount`. On output, each array element contains an area name. The area name in the first element corresponds to the area ID in the first element of the array specified by `areaID`, and so on. If you don't want to get area names, set `areaName` to `NULL`.
function result	A value of `noErr` indicates that `OTCfgGetAreasList` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetAreasList` function obtains the IDs and names of areas in the database and stores this information in two arrays: one containing area IDs and the other containing area names. Each area ID and area name pair identifies an area in the database.

When you allocate the arrays for the `areaID` and `areaName` parameters, be sure to allocate enough elements to hold the number of areas returned by `OTCfgGetAreasCount` (page 68). The actual number of items returned in each array may be lower than the number returned by `OTCfgGetAreasCount` (page 68) if areas have been deleted in the meantime.

OTCfgCreateArea

Creates an area in the Network Setup database.

```
OSStatus OTCfgCreateArea (CfgDatabaseRef dbRef,
                          ConstStr255Param areaName,
                          CfgAreaID* areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaName`	On input, a value of type `ConstStr255Param` that specifies the user-visible name of the area to create. If an area of the name specified by `areaName` already exists, `OTCfgCreateArea` returns `kCfgErrAreaAlreadyExists`.
`areaID`	On input, a pointer to a value of type `CfgAreaID` (page 98). On output, `areaID` contains the ID of the area that was created.
function result	A value of `noErr` indicates that `OTCfgCreateArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgCreateArea` function creates an area of the specified name in the database.

IMPORTANT

The `OTCfgCreateArea` function has almost no purpose in the version of Network Setup described by this document. ◆

OTCfgDuplicateArea

Copies the contents of one area to another area.

```
OSStatus OTCfgDuplicateArea (CfgDatabaseRef dbRef,
                     CfgAreaID sourceAreaID,
                     CfgAreaID destAreaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`sourceAreaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be duplicated. If the area specified by `areaID` does not exist, `OTCfgDuplicateArea` returns `kCfgErrAreaNotFound`.
`destAreaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to contain the duplicated area. If the area specified by `areaID` does not exist, `OTCfgDuplicateArea` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgDuplicateArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgDuplicateArea` function copies the contents of the area specified by `sourceAreaID` into the area specified by `destAreaID`. Both areas must exist before you call `OTCfgDuplicateArea`. To create an area, call `OTCfgCreateArea` (page 70).

IMPORTANT

The `OTCfgDuplicateArea` function has almost no purpose in the version of Network Setup described by this document. ◆

OTCfgDeleteArea

Deletes an area in the Network Setup database.

```
OSStatus OTCfgDeleteArea (CfgDatabaseRef dbRef,
                          CfgAreaID areaID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be deleted. If the area specified by `areaID` does not exist, `OTCfgDeleteArea` returns `kCfgErrAreaNotFound`.
function result	A value of `noErr` indicates that `OTCfgDeleteArea` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgDeleteArea` function removes the specified area from the database.

IMPORTANT

The `OTCfgDeleteArea` function has almost no purpose in the version of Network Setup described by this document. ◆

Managing Entities

Use the following functions to create, modify, and delete entities within an area:

- `OTCfgGetEntitiesCount` (page 73) obtains the number of entities in an area.
- `OTCfgGetEntitiesList` (page 74) obtains a list of entities in an area.
- `OTCfgIsSameEntityRef` (page 76) determines whether two entity references are the same.
- `OTCfgCreateEntity` (page 76) creates an entity in an area.
- `OTCfgDeleteEntity` (page 78) deletes an entity from an area.

- `OTCfgDuplicateEntity` (page 78) copies the contents of one entity to another entity.
- `OTCfgGetEntityLogicalName` (page 79) gets the name of an entity.
- `OTCfgGetEntityName` (page 80) gets the name of an entity.
- `OTCfgSetEntityName` (page 81) sets the name of an entity in an area.
- `OTCfgGetEntityArea` (page 82) gets the area ID of an entity.
- `OTCfgChangeEntityArea` (page 82) changes an entity's area.

OTCfgGetEntitiesCount

Obtains the number of entities of a specified class and type in an area.

```
OSStatus OTCfgGetEntitiesCount (CfgDatabaseRef dbRef,
                     CfgAreaID areaID,
                     CfgEntityClass entityClass,
                     CfgEntityType entityType,
                     ItemCount* itemCount);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be searched. If the area specified by `areaID` does not exist, `OTCfgGetEntitiesCount` returns the error `kCfgErrAreaNotFound`.
`entityClass`	On input, a value of type `CfgEntityClass` that specifies the class that is to be matched. To specify all classes, set `entityClass` to `kCfgClassAnyEntity`. For a list of possible classes, see "Entity Classes and Types" (page 104).
`entityType`	On input, a value of type `CfgEntityType` that specifies the type that is to be matched. To specify all types, set `entityType` to `kCfgTypeAnyEntity`. For a list of possible types, see "Entity Classes and Types" (page 104).

`itemCount`	On input, a pointer to a value of type `ItemCount`. On output, `itemCount` contains the number of entities that matched the specified class and type.
function result	A value of `noErr` indicates that `OTCfgGetEntitiesCount` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetEntitiesCount` function obtains the number of entities of the specified class and type in the specified area. With the number of entities, you can call `OTCfgGetEntitiesList` (page 74) to get the list of entities in the area.

OTCfgGetEntitiesList

Obtains information about entities in an area.

```
OSStatus OTCfgGetEntitiesList (CfgDatabaseRef dbRef,
                     CfgAreaID areaID,
                     CfgEntityClass entityClass,
                     CfgEntityType entityType,
                     ItemCount* itemCount,
                     CfgEntityRef entityRef[],
                     CfgEntityInfo entityInfo[]);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area that is to be searched. If the area specified by `areaID` does not exist, `OTCfgGetEntitiesCount` returns the error `kCfgErrAreaNotFound`.
`entityClass`	On input, a value of type `CfgEntityClass` that specifies the class that is to be matched. To specify all classes, set `entityClass` to `kCfgClassAnyEntity`. For a list of possible classes, see "Entity Classes and Types" (page 104).

`entityType`	On input, a value of type `CfgEntityType` that specifies the type that is to be matched. To specify all types, set `entityType` to `kCfgTypeAnyEntity`. For a list of possible types, see “Entity Classes and Types” (page 104).
`itemCount`	On input, a pointer to a value of type `ItemCount` that specifies the number of entities to list. Call `OTCfgGetEntitiesCount` (page 73) to get the current number of entities in the area represented by `areaID`. On output, `itemCount` points to the number of entities for which information was actually obtained.
`entityRef`	On input, an array of elements of type `CfgEntityRef` (page 99) that is large enough to hold the number of entity references specified by `itemCount`, or `NULL` if you do not want to receive entity references. If not `NULL` on input, each element of the `entityRef` array corresponds to an element of the `entityInfo` array on output.
`entityInfo`	On input, an array of `CfgEntityInfo` (page 99) structures that is large enough to hold the number of `CfgEntityInfo` structures specified by `itemCount`, or `NULL` if you do not want to receive `CfgEntityInfo` structures.
function result	A value of `noErr` indicates that `OTCfgGetEntitiesList` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgGetEntitiesList` function obtains an array of entity references, each of which represents an entity in the specified area, and an array of entity information structures, each of which contains information about its respective entity. The information includes the entity’s class, type, user-visible name, and icon.

You can use the entity reference to call other Network Setup functions, such as `OTCfgOpenPrefs` (page 84).

OTCfgIsSameEntityRef

Compares two entity references.

```
Boolean OTCfgIsSameEntityRef (const CfgEntityRef* entityRef1,
                              const CfgEntityRef* entityRef2,
                              Boolean ignoreArea);
```

`entityRef1`	On input, a pointer to a value of type `CfgEntityRef` (page 99) for one of the entity references that is to be compared.
`entityRef2`	On input, a pointer to a value of type `CfgEntityRef` (page 99) for the second entity reference that is to be compared.
`ignoreArea`	On input, a Boolean value. If `ignoreArea` is `kCfgIgnoreArea`, `OTCfgIsSameEntityRef` ignores the area ID when comparing `entityRef1` and `entityRef2`. If `ignoreArea` is `kCfgDontIgnoreArea`, `OTCfgIsSameEntityRef` does not ignore the area ID when comparing `entityRef1` and `entityRef2`.
function result	`TRUE` if the entity references represent the same entity; `FALSE` if the entity references represent different entities.

DISCUSSION

The `OTCfgIsSameEntityRef` function determines whether two entity references represent the same area. For a discussion of the circumstances in which calling `OTCfgIsSameEntityRef` is particularly useful, see “Areas and Sets” (page 48).

OTCfgCreateEntity

Creates an entity in an area.

```
OSStatus OTCfgCreateEntity (CfgDatabaseRef dbRef,
                            CfgAreaID areaID,
                            CfgEntityInfo* entityInfo,
                            CfgEntityRef* entityRef);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`areaID`	On input, a value of type `CfgAreaID` (page 98) that identifies the area in which the entity is to be created. If the area specified by `areaID` is not writable, `OTCfgCreateEntity` returns the error `kCfgErrLocked`. If the area specified by `areaID` does not exist, `OTCfgCreateEntity` returns the error `kCfgErrAreaNotFound`.
`entityInfo`	On input, a pointer to a `CfgEntityInfo` (page 99) structure that specifies the class, type, user-visible name, and icon for the entity that is to be created. If an entity of the specified name already exists, `OTCfgCreateEntity` returns the error `kCfgErrEntityAlreadyExists`.
`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99). On output, `entityRef` points to an entity reference for the created reference.
function result	A value of `noErr` indicates that `OTCfgCreateEntity` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgCreateEntity` function creates an entity in the area specified by `areaID` with the class, type, user-visible name, and icon specified by the `entityInfo` parameter.

The area represented by `areaID` must have been opened by calling `OTCfgBeginAreaModifications` (page 63).

The `OTCfgCreateEntity` function returns a reference to the created entity that can be passed as a parameter to other Network Setup functions, such as `OTCfgOpenPrefs` (page 84).

OTCfgDeleteEntity

Deletes the specified entity.

```
OSStatus OTCfgDeleteEntity (CfgDatabaseRef dbRef,
                            const CfgEntityRef* entityRef);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) representing the entity that is to be deleted. If `entityRef` represents an entity that does not reside in an area that is open for writing, `OTCfgDeleteEntity` returns the error `kCfgErrAreaNotOpen` or `kCfgErrLocked`. If the entity represented by `entityRef` does not exist, `OTCfgDeleteEntry` returns the error `kCfgErrEntityNotFound`.
function result	A value of `noErr` indicates that `OTCfgDeleteEntity` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgDeleteEntity` function deletes the specified entity.

OTCfgDuplicateEntity

Copies the contents of one entity to another entity.

```
OSStatus OTCfgDuplicateEntity (CfgDatabaseRef dbRef,
                            const CfgEntityRef* entityRef,
                            const CfgEntityRef* newEntityRef);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).

`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that identifies the entity reference that is to be duplicated. If the entity represented by `entityRef` does not exist, `OTCfgDuplicateEntry` returns the error `kCfgErrEntityNotFound`.
`newEntityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that identifies the entity that is to be overwritten by the contents of `entityRef`. If `entityRef` represents an entity that does not reside in an area that is open for writing, `OTCfgDuplicateEntity` returns the error `kCfgErrAreaNotOpen` or `kCfgErrLocked`.
function result	A value of `noErr` indicates that `OTCfgDuplicateEntity` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgDuplicateEntity` function copies the contents of the entity specified by `entityRef` to the entity specified by `newEntityRef`. Any data stored in `newEntityRef` before `OTCfgDuplicateEntity` is called is overwritten by the contents of `entityRef` when `OTCfgDuplicateEntity` returns.

OTCfgGetEntityLogicalName

Obtains the user-visible name of an entity.

```
OSStatus OTCfgGetEntityLogicalName( CfgDatabaseRef dbRef,
                        const CfgEntityRef *entityRef,
                        Str255 entityName );
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that identifies the entity whose name is to be obtained. To obtain the reference for an entity, call `OTCfgGetEntitiesList` (page 74).
`entityName`	On input, a value of type `Str255`. On output, `entityName` contains the user-visible name of the entity represented by `entityRef`.

function result A value of `noErr` indicates that `OTCfgGetEntityLogicalName` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetEntityLogicalName` function obtains the user-visible name of the entity represented by `entityRef`.

Note
The `OTCfgGetEntityLogicalName` function is available in Network Setup 1.2 and later. If OTCfgGetEntityLogicalName is not available, you can get the user-visible name of an entity by calling `OTCfgGetPrefs` (page 86) and specifying `kOTCfgUserVisibleNamePref` as the preference to get. ◆

OTCfgGetEntityName

Obtains the name of an entity.

```
void OTCfgGetEntityName (const CfgEntityRef *entityRef,
                         Str255 entityName);
```

`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that identifies the entity whose name is to be obtained. To obtain the reference for an entity, call `OTCfgGetEntitiesList` (page 74).
`entityName`	On input, a value of type `Str255`. On output, `entityName` contains the name of the entity represented by `entityRef`.
function result	None.

DISCUSSION

The `OTCfgGetEntityName` function obtains the name of the entity represented by `entityRef`.

▲ **WARNING**
The `OTCfgGetEntityName` function does not return the user-visible name of the entity. Instead, `OTCfgGetEntityName` returns an internal name in `entityName`. To get the user-visible name, call `OTCfgGetPrefs` (page 86) passing `kOTCfgUserVisibleNamePref` in the `prefsType` parameter or call `OTCfgGetEntityLogicalName` (page 79) if that function is available. ▲

OTCfgSetEntityName

Sets the user-visible name of an entity.

```
OSStatus OTCfgSetEntityName (CfgDatabaseRef dbRef,
                         const CfgEntityRef* entityRef,
                         ConstStr255Param entityName,
                         CfgEntityRef* newEntityRef);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that represents the entity whose name is to be set. To obtain the entity reference for an entity, call `OTCfgGetEntitiesList` (page 74). If `entityRef` does not refer to a valid entity, `OTCfgSetEntityName` returns the error `kCfgEntityNotFoundErr`.
`entityName`	On input, a value of type `ConstStr255Param` that specifies the new user-visible name for the entity.
`newEntityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99). On output, `newEntityRef` points to a new entity reference that represents the renamed entity. Your application should use `newEntityRef` for future references to the renamed entity.
function result	A value of `noErr` indicates that `OTCfgSetEntityName` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgSetEntityName` function sets the user-visible name of the specified entity and returns a new entity reference for the renamed entity.

OTCfgGetEntityArea

Obtains the area ID of an entity.

```
void OTCfgGetEntityArea (const CfgEntityRef *entityRef,
                         CfgAreaID *areaID);
```

`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that identifies the entity reference whose area is to be obtained. To obtain the entity reference for an entity, call `OTCfgGetEntitiesList` (page 74).
`areaID`	On input, a pointer to a value of type `CfgAreaID` (page 98). On output, `areaID` points to the area ID of the entity represented by `entityRef`.
function result	None.

DISCUSSION

The `OTCfgGetEntityArea` function obtains the area ID of the entity represented by `entityRef`.

OTCfgChangeEntityArea

Changes the area of an entity.

```
void OTCfgChangeEntityArea (CfgEntityRef *entityRef,
                         CfgAreaID newAreaID);
```

`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that represents the entity reference whose area is to be changed. To obtain the entity reference for an entity, call `OTCfgGetEntitiesList` (page 74) or use the entity reference returned by a Network Setup function that creates an entity.
`newAreaID`	On input, a value of type `CfgAreaID` (page 98) that specifies the new area ID for the specified entity.
function result	None.

DISCUSSION

The `OTCfgChangeEntityArea` function changes the area ID of the specified entity. This function does not actually move the entity. Instead, it changes the entity reference to point to the same entity in the area specified by `newAreaID`.

Managing Preferences

Use the following functions to manage preferences, which are stored in an entity:

- `OTCfgOpenPrefs` (page 84) opens an entity so that its preferences can be accessed.
- `OTCfgClosePrefs` (page 85) closes an entity.
- `OTCfgGetPrefsSize` (page 85) gets the size of a preference.
- `OTCfgGetPrefs` (page 86) gets the value of a preference.
- `OTCfgSetPrefs` (page 87) sets the value of a preference.
- `OTCfgGetPrefsTOCCount` (page 88) gets the number of preferences in an entity.
- `OTCfgGetPrefsTOC` (page 89) gets a list of a preferences in an entity.
- `OTCfgGetDefault` (page 90) gets the default value for a preference.
- `OTCfgDeletePrefs` (page 90) deletes a preference from an entity.
- `OTCfgGetTemplate` (page 91) gets a preference's template.

OTCfgOpenPrefs

Opens an entity so that its preferences can be accessed.

```
OSStatus OTCCfgOpenPrefs (CfgDatabaseRef dbRef,
                          const CfgEntityRef* entityRef,
                          Boolean writer,
                          CfgEntityAccessID* accessID);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`entityRef`	On input, a pointer to a value of type `CfgEntityRef` (page 99) that represents the entity whose preferences are to be read or written. If the entity does not exist, `OTCfgOpenPrefs` returns the error `kCfgErrEntityNotFound`.
`writer`	On input, a Boolean value. If `writer` is `TRUE`, the entity represented by `entityRef` must be in an area that was opened by calling `OTCfgBeginAreaModifications` (page 63); otherwise, `OTCfgOpenPrefs` returns the error `kCfgErrLocked`. If writer is `FALSE`, the entity represented by `entityRef` must be in an open area [opened by calling `OTCfgBeginAreaModifications` (page 63) or `OTCfgOpenArea` (page 61)]; otherwise, `OTCfgOpenPrefs` returns the error `kCfgErrAreaNotOpen`.
`accessID`	On input, a pointer to a value of type `CfgEntityAccessID` (page 100). On output, use `accessID` in subsequent calls to get and set preferences.
function result	A value of `noErr` indicates that `OTCfgOpenPrefs` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgOpenPrefs` function opens the specified entity so that your application can get or set the value of the preferences the entity contains.

If the value of the `writer` parameter is `TRUE`, you can set preferences as well as get preferences; otherwise, you can only get preferences.

OTCfgClosePrefs

Closes an entity.

```
OSStatus OTCCfgClosePrefs (CfgEntityAccessID accessID);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84), that identifies the entity that is to be closed.
function result	A value of `noErr` indicates that `OTCfgClosePrefs` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgClosePrefs` function closes the specified entity.

OTCfgGetPrefsSize

Gets the size of a preference.

```
OSStatus OTCCfgGetPrefsSize (CfgEntityAccessID accessID.
                     OSType prefsType,
                     ByteCount * length);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84), that identifies the entity containing the preference whose size is to be obtained.
`prefsType`	On input, a value of type `prefsType` that identifies the type of the preference whose size is to be obtained.
`length`	On input, a pointer to a value of type `ByteCount`. On output, `length` contains the size in bytes of the preference specified by `prefsType`.

function result	A value of `noErr` indicates that `OTCfgGetPrefsSize` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetPrefsSize` function gets the size in bytes of the preference specified by `prefsType` in the entity represented by `accessID`.

For variable-length preferences, you should call `OTCfgGetPrefsSize` to get the size of a preference before it calls `OTCfgGetPrefs` (page 86) to get the value of that preference.

OTCfgGetPrefs

Gets the value of a preference.

```
OSStatus OTCCfgGetPrefs (CfgEntityAccessID accessID.
                         OSType prefsType,
                         void* data,
                         ByteCount length);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84), that identifies the entity containing the preference whose value is to be obtained.
`prefsType`	On input, a value of type `OSType` that identifies the preference whose value is to be obtained. See"Protocol Constants and Other Data Types" (page 159) for protocol-specific preferences.
`data`	On input, a pointer to the buffer into which the value of the preference is to be placed. On output, `data` contains the value of the preference specified by `prefsType`.
`length`	On input, a value of type `ByteCount` that is the size in bytes of the buffer pointed to by `data`.
function result	A value of `noErr` indicates that `OTCfgGetPrefs` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The OTCfgGetPrefs function gets the value of the preference specified by prefsType in the entity represented by accessID and stores it in data.

Before calling OTCfgGetPrefs, you may call OTCfgGetPrefsSize (page 85) to obtain the size of the entity so that you can allocate a data parameter of the appropriate size.

If the data parameter is too small to hold the value, OTCfgGetPrefs stores as much of the value in data as possible and returns the error kCFGErrDataTruncated.

OTCfgSetPrefs

Sets the value of a preference.

```
OSStatus OTCCfgSetPrefs (CfgEntityAccessID accessID.
                         OSType prefsType,
                         const void* data,
                         ByteCount length);
```

accessID	On input, a value of type CfgEntityAccessID (page 100), obtained by previously calling OTCfgOpenPrefs (page 84). The entity in which the preference represented by accessID resides must itself reside in an area that has been opened for writing by calling OTCfgBeginAreaModifications (page 63).
prefsType	On input, a value of type OSType that identifies the preference to set. If a preference of the type specified by prefsType already exists OTCfgSetPrefs overwrites the value of the preference. Otherwise, OTCfgSetPrefs creates the new preference.
data	On input, a pointer to the data that is to be set.
length	On input, a value of type ByteCount that contains the length in bytes of the data in data.
function result	A value of noErr indicates that OTCfgSetPrefs returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgSetPrefs` function sets the preference represented by `prefsType` to the value specified by `data`.

The `accessID` parameter must have been created by calling `OTCfgOpenPrefs` (page 84) with the `writer` parameter set to `TRUE`; otherwise, `OTCfgSetPrefs` returns the error `kCfgErrLocked`.

OTCfgGetPrefsTOCCount

Gets the number of preferences in an entity.

```
OSStatus OTCfgGetPrefsTOCCount (CfgEntityAccessID accessID.
                        ItemCount *itemCount);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84) that identifies the entity whose preferences are to be counted.
`itemCount`	On input, a pointer to a value of type `ItemCount`. On output, `itemCount` contains the number of preferences in the entity represented by `accessID`.
function result	A value of `noErr` indicates that `OTCfgGetPrefsTOCCount` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgGetPrefsTOCCount` function gets the number of preferences in the entity represented by `accessID`.

You should call `OTCfgPrefsTOCCount` to find out how many preferences are present before calling `OTCfgGetPrefsTOC` (page 89).

OTCfgGetPrefsTOC

Gets a list of the preferences in an entity.

```
OSStatus OTCfgGetPrefsTOC (CfgEntityAccessID accessID.
                          ItemCount* itemCount,
                          CfgPrefsHeader prefsTOC[]);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84) that identifies the entity whose preferences are to be obtained.
`itemCount`	On input, a pointer to a value of type `ItemCount` that specifies the requested number of preferences. On output, `itemCount` contains the number of preferences that were obtained.
`prefsTOC`	On input, an array of `CfgPrefsHeader` (page 100) structures. The `prefsTOC` parameter must have enough `CfgPrefsHeader` structures to store all of the preferences in the entity.
function result	A value of `noErr` indicates that `OTCfgGetPrefsTOC` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetPrefsTOC` function obtains information about the specified number preferences in the entity represented by `accessID` and stores them in the `prefsTOC` array.

Before you call `OTCfgPrefsTOC`, you must should find out how many preferences are available by calling `OTCfgGetPrefsTOCCount` (page 88).

▲ **WARNING**
Early versions of Network Setup do not determine whether there is enough space in `prefsTOC` (as specified on input by `itemCount`) and can write beyond the end of the array. You should always call `OTCfgGetPrefsTOCCount` before calling `OTCfgGetPrefsTOC`. When you call `OTCfgGetPrefsTOC`, set `itemCount` to the value returned by `OTCfgGetPrefsTOCCount` in the `itemCount` parameter. ▲

OTCfgGetDefault

Returns a handle containing the default value for a preference.

```
Handle OTCfgGetDefault (OSType entityType,
                        OSType entityClass,
                        OSType prefsType);
```

`entityType`	On input, a value of type `OSType` that identifies the entity type of the default preference that is to be obtained. For possible values, see "Entity Classes and Types" (page 104).
`entityClass`	On input, a value of type `OSType` that identifies the entity class of the default preference that is to be obtained. For possible values, see "Entity Classes and Types" (page 104).
`prefsType`	On input, a value of type `OSType` that identifies the preference whose default value is to be obtained.
function result	A handle or `NULL` if no preference of the specified entity type, class, and preference type exists, or if there is not enough memory to obtain the handle.

DISCUSSION

The `OTCfgGetDefault` function returns a handle containing the default value for a preference of the specified entity, class, and preference type.

Note
You are responsible for disposing of the handle that `OTCfgGetDefault` obtains by calling the Memory Manager function `DisposeHandle`. ◆

OTCfgDeletePrefs

Deletes a preference.

```
OSStatus OTCfgDeletePrefs (CfgEntityAccessID accessID,
                           OSType prefsType);
```

`accessID`	On input, a value of type `CfgEntityAccessID` (page 100), obtained by previously calling `OTCfgOpenPrefs` (page 84) that identifies the entity from which a preference is to be deleted.
`prefsType`	On input, a value of type `OSType` that identifies the preference type of the preference that is to be deleted.
function result	A value of `noErr` indicates that `OTCfgDeletePrefs` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgDeletePrefs` function deletes the preference of the type specified by `prefsType` from the entity specified by `accessID`.

Note
The `OTCfgDeletePrefs` function is available in Network Setup version 1.2 and later.

OTCfgGetTemplate

Gets the default value for a specific preference.

```
OSStatus OTCfgGetTemplate(CfgEntityClass entityClass,
                          CfgEntityType  entityType,
                          OSType prefsType,
                          void *data,
                          ByteCount *dataSize);
```

`entityClass`	On input, a value of type `CfgEntityClass` that specifies the class of the preference whose default value is to be obtained.
`entityType`	On input, a value of type `CfgEntityType` that specifies the type of the preference whose default value is to be obtained.
`prefsType`	On input, a value of type `OSType` that specifies the preference type of the preference whose default value is to be obtained.

`data`	On input, a pointer to the buffer into which the default value is to be placed. On output, `data` points to the default value. If the buffer is too small to hold the default value, `OTCfgGetTemplate` returns as much data as possible and returns the error `kCFGErrDataTruncated`. If you want to get the size of the default value but not the default value itself, set data to `NULL`.
`dataSize`	On input, a pointer to a value of type `ByteCount`. On output, `dataSize` points to the number of bytes in the buffer pointed to by `data`. On input, if `data` is `NULL`, on output, `dataSize` points to the size in bytes of the default value for the specified preference.
function result	A value of `noErr` indicates that `OTCfgGetTemplate` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110).

DISCUSSION

The `OTCfgGetTemplate` function gets the default value for the preference identified by the `entityClass`, `entityType`, and `prefsType` parameters and stores it in the buffer described by `data` and `dataSize`.

IMPORTANT

The `OTCfgGetTemplate` function is available in Network Setup version 1.2 and later. It returns the same data that `OTCfgGetDefault` (page 90) returns, but the parameters have been changed to be consistent with the parameters of other Network Setup functions. If you rely on Network Setup 1.2 or later, call `OTCfgGetTemplate`. If you need to work with earlier versions of Network Setup, you can safely continue to call `OTCfgGetDefault`. ▲

Preference Utilities

Use the following functions to encrypt and decrypt preferences:

- `OTCfgDecrypt` (page 93) decrypts data.
- `OTCfgEncrypt` (page 93) encrypts data.

OTCfgEncrypt

Encrypts data.

```
SInt16 OTCCfgEncrypt (const UInt8 *key.
                      UInt8 *data,
                      SInt16 dataLen);
```

`key`	On input, a pointer to a Pascal string containing the encryption key. For Remote Access password, the encryption key is a user name.
`data`	On input, a pointer to an array of bytes that contains data that is to be encrypted. Usually, the data is a password. On output, `data` contains the encrypted password.
`dataLen`	On input, a value of type `SInt16` that specifies the number of bytes in the data array.
function result	The length of the encrypted data.

DISCUSSION

The `OTCfgEncrypt` function encrypts the contents of the `data` parameter using the key specified by the `key` parameter. For sample code, see Listing 2-17 in Chapter 2, "Using Network Setup."

Note
The `OTCCfgEncrypt` function is available in Network Setup version 1.1 and later. ◆

OTCfgDecrypt

Decrypts data.

```
SInt16 OTCCfgDecrypt (const UInt8 *key.
                      UInt8 *data,
                      SInt16 dataLen);
```

`key`	On input, a pointer to a Pascal string containing the encryption key. Usually the encryption key is a user name.
`data`	On input, a pointer to an array of bytes containing data that was previously encrypted by `OTCfgEncrypt` (page 93). On output, `data` contains the decrypted data.
`dataLen`	On input, a value of type `SInt16` that specifies the length of `data`.
function result	The length in bytes of the decrypted data.

DISCUSSION

The `OTCfgDecrypt` function decrypts the contents of the `data` parameter using the key specified by the `key` parameter.

Note
The `OTCCfgDecrypt` function is available in Network Setup version 1.1 and later. ◆

Installing and Removing a Notification Callback

You can use the following functions to install and remove a notification callback:

- `OTCfgInstallNotifier` (page 94) installs a notification callback.
- `OTCfgRemoveNotifier` (page 96) removes a notification callback.

OTCfgInstallNotifier

Installs a notification callback.

```
OSStatus OTCfgInstallNotifier (CfgDatabaseRef dbRef.
                    CfgEntityClass theClass,
                    CfgEntityType theType,
                    OTNotifyProcPtr notifier,
                    void* contextPtr);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`theClass`	On input, a value of type `CfgEntityClass` that specifies the class for which the notification callback is to be called. For possible values, see the constants described in "Entity Classes and Types" (page 104). Constants that define wildcards are valid.
`theType`	On input, a value of type `CfgEntityType` that specifies the type for which the notification callback is to be called. For possible values, see the constants described in "Entity Classes and Types" (page 104). Constants that define wildcards are valid.
`notifier`	On input, a value of type `OTNotifyProcPtr` that points to the notification callback that is to be installed.
`contextPtr`	On input, a pointer to an arbitrary data type that is passed to the notification callback when it is called.
function result	A value of `noErr` indicates that `OTCfgInstallNotifier` returned successfully. For a list of other possible result codes, see "Result Codes" (page 110)

DISCUSSION

The `OTCfgInstallNotifier` function installs a notification callback that is called when changes to preferences of the specified class and type occur. Calling `OTCfgInstallNotifier` when you have already installed a notification callback causes the current notification callback to be replaced by the new notification callback.

Note
The `OTCfgInstallNotifier` function is available in Network Setup version 1.0.2 and later. ◆

To remove an installed notification callback, call `OTCfgRemoveNotifier` (page 96). Notification callbacks are removed automatically when the database session represented by `dbRef` is closed.

OTCfgRemoveNotifier

Removes a notification callback.

```
OSStatus OTCfgRemoveNotifier (CfgDatabaseRef dbRef.
                        CfgEntityClass theClass,
                        CfgEntityType theType);
```

`dbRef`	On input, a value of type `CfgDatabaseRef` (page 98) that represents a database session previously opened by calling `OTCfgOpenDatabase` (page 58).
`theClass`	On input, a value of type `CfgEntityClass` specifying the class that was specified when the notification callback was installed.
`theType`	On input, a value of type `CfgEntityType` specifying the type that was specified when the notification callback was installed.
function result	A value of `noErr` indicates that `OTCfgRemoveNotifier` returned successfully. For a list of other possible result codes, see “Result Codes” (page 110).

DISCUSSION

The `OTCfgRemoveNotifier` function removes the specified notification callback that was previously installed by `OTCfgInstallNotifier` (page 94).

Note
The `OTCfgRemoveNotifier` function is available in Network Setup version 1.0.2 and later. ◆

Notification callbacks are removed automatically when the database session represented by `dbRef` is closed.

Application-Defined Routines

This section describes the application-defined routine that you can provide:

- A notification callback routine, which is called when changes occur in the Network Setup database.

Notification Callback Routine

Receives notifications of changes to the Network Setup database.

```
typedef CALLBACK_API_C( void, OTNotifyProcPtr ) (
                     void *contextPtr,
                     OTEventCode code,
                     OSStatus result,
                     void *cookie);
```

`contextPtr`	A pointer to the untyped value that was specified when you called `OTCfgInstallNotifier` (page 94) to install the notification callback routine.
`code`	A value of type `OTEventCode`.Your notification callback routine should ignore callbacks when this is any value other than `kCfgDatabaseChanged`.
`result`	A notification-dependent value of type `OSStatus`. When the value of code is `kCfgDatabaseChanged`, the value of result is `kCfgErrDatabaseChanged`.
`cookie`	Reserved.

DISCUSSION

Your notification callback routine is called at system task time (but not necessarily in the context of your application) when a change occurs to the database. When your notification callback routine is called, you should reread any preferences that were previously read.

Network Setup Structures and Data Types

This section describes structures used by the Network Setup functions. The structures and data types are

- `CfgDatabaseRef` (page 98), which refers to an open database session.
- `CfgAreaID` (page 98), which identifies an area.
- `CfgEntityRef` (page 99), which refers to an open entity.

- CfgEntityInfo (page 99), which contains information about the entities in an area.
- CfgEntityAccessID (page 100), which identifies an open preference within an entity.
- CfgPrefsHeader (page 100), which is used to return information about the preferences within an entity.
- CfgSetsStruct (page 101), which stores information about a set entity.
- CfgSetsElement (page 103), which represents an element in a CfgSetsVector (page 103) structure.
- CfgSetsVector (page 103), which stores references to a set of entities.

CfgDatabaseRef

A value of type CfgDatabaseRef refers to an open session with the Network Setup database.

```
typedef struct OpaqueCfgDatabaseRef* CfgDatabaseRef;
```

CfgDatabaseRef	A pointer to an opaque value that identifies the open session.

Call OTCfgOpenDatabase (page 58) to open the database and obtain a value of type CfgDatabaseRef. Network Setup requires a value of type CfgDatabaseRef to open an area, make changes in an area, list and create entities in an area, and to open an entity.

A CfgDatabaseRef whose value is NULL is never a valid database reference.

CfgAreaID

A value of type CfgAreaID identifies an area.

```
typedef UInt32 CfgAreaID;
```

CfgAreaID	An unsigned 32-bit value that uniquely identifies an area.

Network Setup uses a value of type `CfgAreaID` to identify the area in which an entity resides. For example, a value of type `CfgAreaID` is a member of the `CfgEntityRef` (page 99) structure. Use the constant `kInvalidCfgAreaID` (page 109) to determine whether an area ID is valid.

CfgEntityRef

A `CfgEntityRef` structure refers to a specific entity.

```
struct CfgEntityRef {
    CfgAreaID fLoc;
    UInt32 fReserved;
    Str255 fID;
};
typedef struct CfgEntityRef CfgEntityRef;
```

Field descriptions

`fLoc`	The area in which the entity resides.
`fReserved`	Reserved.
`fID`	The entity ID.

For example, `OTCfgCreateEntity` (page 76) returns a `CfgEntityRef` structure to refer to the newly created entity, and `OTCfgGetEntitiesList` (page 74) returns a `CfgEntityRef` for each entity in an area.

CfgEntityInfo

The `CfgEntityInfo` structure stores various attributes of an entity.

```
struct CfgEntityInfo {
    CfgEntityClass fClass;
    CfgEntityType fType;
    Str255 fName;
    CfgResourceLocator fIcon;
};
typedef struct CfgEntityInfo CfgEntityInfo;
```

Field descriptions

`fClass`	The entity's class. See "Entity Classes and Types" (page 104) for possible values.
`fType`	The entity's type. See "Entity Classes and Types" (page 104) for possible values.
`fName`	The entity's user-visible name.
`fIcon`	The entity's custom icon. For details, see the definition of `CfgResourceLocator` (page 101).

`CfgEntityInfo` structures are used when calling `OTCfgCreateEntity` (page 76) and when calling `OTCfgGetEntitiesList` (page 74).

CfgEntityAccessID

A `CfgEntityAccessID` refers to an open preference.

```
typedef void *CfgEntityAccessID;
```

`CfgEntityAccessID`	A pointer to an arbitrary data type whose value represents an open entity

Call `OTCfgOpenPrefs` (page 84) to open an entity and received a value of type `CfgEntityAccessID`. Pass `CfgEntityAccessID` as a parameter to `OTCfgGetPrefsSize` (page 85) and then `OTCfgGetPrefs` (page 86) to get the value of a preference and to `OTCfgSetPrefs` (page 87) to set its value.

A `CfgEntityAccessID` whose value is `NULL` is never a valid entity access ID.

CfgPrefsHeader

The `CfgPrefsHeader` structure is used to return information about preferences in an entity.

```
struct CfgPrefsHeader {
    UInt16  fSize;
    UInt16  fVersion;
    OSType  fType;
};
typedef struct CfgPrefsHeader CfgPrefsHeader;
```

Field descriptions

`fSize`	The size in bytes of the preference, not including the `CfgPrefsHeader` structure itself.
`fVersion`	Always zero in the version of Network Setup described by this document.
`fType`	An OS type that uniquely identifies the preference within the entity.

To get the `CfgPrefsHeader` structures for an entity, call `OTCfgGetPrefsTOC` (page 89).

CfgResourceLocator

The `CfgResourceLocator` structure contains a file specification and a resource ID for an entity's custom icon.

```
struct CfgResourceLocator {
    FSSpec      fFile;
    SInt16      fResID;
};
```

Field descriptions

`fFile`	A file specification.
`fResID`	A resource ID.

The `CfgResourceLocator` structure is a member of the `CfgEntityInfo` (page 99) structure. Custom icons are currently not displayed, so you should initialize this structure to zero for any entities that you create.

CfgSetsStruct

The `CfgSetsStruct` structure holds information about a set entity.

```
struct CfgSetsStruct
{
    UInt32 fFlags;
    UInt32 fTimes[ kOTCfgIndexSetsLimit ];
};
typedef struct CfgSetsStruct CfgSetsStruct;
```

Field descriptions

`fFlags`	Flags for this set. For possible values, see the enumeration for the `fFlags` field that follows.
`fTimes`	An array of time stamps used during legacy import and export indexed by the enumeration for the `fTimes` field that follows.

The following enumerations define bits and masks for the `fFlags` field:

```
enum {
    kOTCfgSetsFlagActiveBit = 0
};

enum {
    kOTCfgSetsFlagActiveMask = 0x0001
};
```

If the active bit is set, this set entity is the active set. The default area must always contain exactly one active set.

The following enumeration defines values for the `fTimes` field:

```
enum {
    kOTCfgIndexSetsActive = 0,
    kOTCfgIndexSetsEdit,
    kOTCfgIndexSetsLimit
};
```

Constant descriptions

`kOTCfgIndexSetsActive`	This index yields the time stamp of the active legacy preferences file.
`kOTCfgIndexSetsEdit`	Sets edit index.
`kOTCfgIndexSetsLimit`	This value is defined to allow the declaration of the `fTimes` field of the `CfgSetsStruct` (page 101) structure.

The preference type for the `CfgSetsStruct` structure is `kOTCfgSetsStructPref`, which is defined as `'stru'`.

CfgSetsElement

The `CfgSetsElement` structure represents an element in a `CfgSetsVector` structure.

```
struct CfgSetsElement {
    CfgEntityRef fEntityRef;
    CfgEntityInfo fEntityInfo;
};
typedef struct CfgSetsElement CfgSetsElement;
```

Field descriptions

`fEntityRef`	An entity reference for the entity to be included in this set.
`fEntityInfo`	A `CfgEntityInfo` (page 99) structure that describes the entity referenced by `fEntityRef`.

The `fEntityRef` entity typically has an area ID that doesn't match the area of the set entity. See the section "Areas and Sets" (page 48) for why this happens and how you can work around the mismatch.

CfgSetsVector

The `CfgSetsVector` structure holds references to a set of entities.

```
struct CfgSetsVector
{
    UInt32 fCount;
    CfgSetsElement fElements[1];
};
typedef struct CfgSetsVector CfgSetsVector;
```

Field descriptions

`fCount`	The number of elements in the set.

`fElements`	An unbounded array consisting of the number of `CfgSetsElement` (page 103) structures specified by `fCount`. All of the entities in this array are considered to be part of the set.

The preference type for the `CfgSetsVector` structure is `kOTCfgSetsVectorPref`, which is defined as `'vect'`.

Network Setup Constants

The following sections describe the Network Setup constants:

- "Entity Classes and Types" (page 104)
- "Common Preference Types" (page 106)

Entity Classes and Types

Network Setup can distinguish between several classes of entities and several types within each class. Using classes allows you to store different types of information in the same database. Third-party developers can define additional entity classes and types. If you define an entity class or type, it should be unique and registered with Developer Technical Support (DTS).

The following enumeration defines constants for the classes and types for the entities defined by Apple Computer:

```
enum {
    kOTCfgClassNetworkConnection    = 'otnc',
    kOTCfgClassGlobalSettings       = 'otgl',
    kOTCfgClassServer               = ‘otsv’,
    kOTCfgTypeGeneric               = 'otan',
    kOTCfgTypeAppleTalk             = 'atlk',
    kOTCfgTypeTCPv4                 = 'tcp4',
    kOTCfgTypeTCPv6                 = 'tcp6',
    kOTCfgTypeDNS                   = ‘dns ‘,
    kOTCfgTypeRemote                = 'ara ',
    kOTCfgTypeDial                  = 'dial',
    kOTCfgTypeModem                 = 'modm',
    kOTCfgTypeInfrared              = 'infr',
    kOTCfgClassSetOfSettings        = 'otsc',
    kOTCfgTypeSetOfSettings         = 'otst',
};
```

Constant descriptions

`kOTCfgClassNetworkConnection`	The class code for network connection entities.
`kOTCfgClassGlobalSettings`	The class code global protocol entities.
`kOTCfgClassServer`	The class code for server setting entities.
`kOTCfgTypeGeneric`	The type code for non-specific entities.
`kOTCfgTypeAppleTalk`	The type code for AppleTalk entities.
`kOTCfgTypeTCPv4`	The type code for version 4 of the Transmission Control Protocol/Internet Protocol (TCP/IP) entities.
`kOTCfgTypeTCPv6`	The type code for TCP/IP version 6 entities.
`kOTCfgTypeRemote`	The type code for Apple Remote Access (ARA) entities.
`kOTCfgTypeDial`	The type code for Dial Assist entities.
`kOTCfgTypeModem`	The type code for Modem entities.
`kOTCfgTypeInfrared`	The type code for Infrared entities.
`kOTCfgClassOfSettings`	The class code for set entities.
`kOTCfgSetOfSettings`	The type code for set entities.
`kOTCfgTypeDNS`	The type code for Domain Name System (DNS) entities.

Wildcard Classes and Types

The following enumeration defines wildcard values for matching or not matching entity classes and entity types:

```
enum {
    kCfgClassAnyEntity              = '****',
    kCfgClassUnknownEntity          = '????',
    kCfgTypeAnyEntity               = '****',
    kCfgTypeUnknownEntity           = '????'
};
```

Constant descriptions

`kCfgClassAnyEntity`	Matches the class type for any entity. This constant is typically used when calling `OTCfgGetEntitiesCount` (page 73) and `OTCfgGetEntitiesList` (page 74).
`kCfgClassUnknownEntity`	Does not match the class type for any entity. Use this constant as a "NULL" equivalent.
`kCfgTypeAnyEntity`	Matches the type for any entity. This constant is typically used when calling `OTCfgGetEntitiesCount` (page 73) and `OTCfgGetEntitiesList` (page 74).
`kCfgTypeUnknownEntity`	Does not match the entity type for any entity. Use this constant as a "NULL" equivalent.

Common Preference Types

This section describes preferences that are used by many different protocols. For protocol-specific preferences, see Chapter 4, "Network Setup Protocol Structures and Data Types."

Per-connection Preference Types

The following enumeration defines per-connection preference types:

```
enum
{
    kOTCfgUserVisibleNamePref            = 'pnam',
    kOTCfgVersionPref                    = 'cvrs',
    kOTCfgPortUserVisibleNamePref        = 'port',
    kOTCfgProtocolUserVisibleNamePref    = 'prot',
    kOTCfgAdminPasswordPref              = 'pwrd',
    kOTCfgProtocolOptionsPref            = 'opts',
    kCfgFreePref                         = ‘free’
};
```

Constant descriptions

`kOTCfgUserVisibleNamePref`
: Each connection entity has a preference of this type that contains the user-visible name of the entity as a Pascal string

`kOTCfgVersionPref`	Some protocols store the version of the protocol in this preference. Typically, this preference is a `UInt16` whose value is 1.
`kOTCfgPortUserVisibleNamePref`	Some protocols use this preference to store the user-visible name of the port over which the protocol is running as a Pascal string.
`kOTCfgProtocoltUserVisibleNamePref`	Some protocols store a user-visible description of the protocol in this preference as a C string. For TCP/IP the value of this preference is "tcp". For AppleTalk, the value of this preference is "ddp".
`kOTCfgAdminPasswordPref`	This preference is not documented.
`kOTCfgProtocolOptionsPref`	Many protocols use this preference (a `UInt32`) to store protocol-specific flags.
`kCfgFreePref`	A dummy preference type used for free blocks in an entity.

Global Preference Types

The following enumeration defines global preference types:

```
enum
{
    kOTCfgUserModePref              = 'ulvl',
    kOTCfgPrefWindowPositionPref    = 'wpos',
};
```

Constant descriptions

`kOTCfgUserModePref`	Preference type for the user mode preference for TCP/IP and AppleTalk only.
`kOTCfgPrefWindowPositionPref`	Preference type for the location (in global coordinates) of the control panel window for TCP/IP, AppleTalk, and Infrared.

Set Entity Preference Types

The following enumeration defines preference types for set entities:

```
enum {
    kOTCfgSetsStructPref= 'stru',
    kOTCfgSetsVectorPref= 'vect',
};
```

Constant descriptions

`kOTCfgSetsStructPref`Preference type for the `CfgSetsStruct` (page 101) structure.

`kOTCfgSetsVectorPref`Preference type for the `CfgSetsVector` (page 103) structure.

Backward Compatibility Preference Types

The following enumeration defines per-connection backward compatibility preference types:

```
enum
{
    kOTCfgCompatNamePref            = 'cnam',
    kOTCfgCompatResourceNamePref    = 'resn',
};
```

These preferences are used by the Network Setup backward compatibility mechanism to ensure an accurate conversion between legacy preference files and the Network Setup database.

Global Backward Compatibility Preference Types

The following enumeration defines global backward compatibility preference types:

```
enum
{
    kOTCfgCompatSelectedPref        = 'ccfg',
    kOTCfgCompatResourceIDPref      = 'resi',
};
```

These preferences are used by the Network Setup backward compatibility mechanism to ensure an accurate conversion between legacy preference files and the Network Setup database.

OTCfgUserMode Preference

For most control panels that support a concept of "user mode," the `OTCfgUserMode` preference holds (or is used as a field in another preference to hold) the current user mode as a `UInt16`. The exceptions are the ARA and Modem control panels, where the user mode is stored as a `UInt32`.

```
enum unsigned short OTCfgUserMode
{
    kOTCfgBasicUserMode     = 1,
    kOTCfgAdvancedUserMode  = 2,
    kOTCfgAdminUserMode     = 3,
};
typedef UInt32 OTCfgUserMode32;
```

Constant descriptions

`kOTCfgBasicUserMode`
: Basic user mode.

`kOTCfgAdvancedUserMode`
: Advanced user mode.

`kOTCfgAdminUserMode`
: Administration user mode. This mode is used by the control panel at runtime but is never valid in a preference. It is defined here for completeness only.

Invalid Area ID

The constant `kInvalidCfgAreaID` represents an invalid area ID.

Result Codes

The result codes specific to Network Setup are listed here. Network Setup functions can also return system error codes, which do not appear in this list.

`kCfgErrDatabaseChanged`	–3290	The database has changed since the last call. Close and reopen the database.
`kCfgErrAreaNotFound`	–3291	The specified area does not exist.
`kCfgErrAreaAlreadyExists`	–3292	The specified area already exists.
`kCfgErrAreaNotOpen`	–3293	The specified area is not open.
`kCfgErrConfigLocked`	–3294	The specified area is locked. Try again later.
`kCfgErrEntityNotFound`	–3295	An entity of the specified name does not exist.
`kCfgErrEntityAlreadyExists`	–3296	An entity of the specified name already exists.
`kCfgErrPrefsTypeNotFound`	–3297	A record of the specified type does not exist.
`kCfgErrDataTruncated`	–3298	Data was truncated because the read buffer is too small.
`kCfgErrFileCorrupted`	–3299	The database is corrupted.

C H A P T E R 4

Network Setup Protocol Structures and Data Types

This chapter describes the structures and data types for protocols provided by Apple Computer.

Protocol Structures

This section describes the structures that organize the information in the Network Setup database.

- The section "TCP/IP Structures" (page 111) describes the structures used by TCP/IP preferences.
- The section "Apple Remote Access Structures" (page 122) describes the structures used by Apple Remote Access (ARA) preferences.
- The section "Modem Structures" (page 141) describes the structures used by modem preferences.
- The section "AppleTalk Structures" (page 144) describes the structures used by AppleTalk preferences.
- The section "Infrared Structures" (page 158) describes the structures used by Infrared preferences.

TCP/IP Structures

This section describes structures that store TCP/IP preferences. The structures are

- `OTCfgTCPInterfacesUnpacked` (page 112) stores information about the configured TCP/IP interfaces in unpacked format.

- `OTCfgTCPInterfacesPacked` (page 114) stores information about the configured TCP/IP interfaces in packed format.
- `OTCfgTCPInterfacesPackedPart` (page 114) is a member of the `OTCfgTCPInterfacesPacked` (page 114) structure that stores port, module, and framing information for TCP/IP interfaces in packed format.
- `OTCfgTCPDHCPLeaseInfo` (page 115) stores information about a DHCP lease.
- `OTCfgTCPDNSServersList` (page 116) stores name server information.
- `OTCfgTCPLocks` (page 116) stores information about whether a preference is locked.
- `OTCfgTCPRoutersList` (page 119) stores an array of `OTCfgTCPRoutersListEntry` (page 119) structures.
- `OTCfgTCPRoutersListEntry` (page 119) stores the IP address of the router that has been configured for this interface as the default gateway.
- `OTCfgTCPSearchDomains` (page 120) stores the list of domains that are searched after the implicit search domains.
- `OTCfgTCPSearchList` (page 120) stores DNS configuration information.
- `OTCfgTCPUnloadAttr` (page 121) defines values that indicate when TCP/IP is loaded.

OTCfgTCPInterfacesUnpacked

The `OTCfgTCPInterfacesUnpacked` structure stores information about the configured TCP/IP interfaces in unpacked format. See Listing 2-16 in Chapter 2, "Using Network Setup," for sample code that packs and unpacks this structure.

IMPORTANT

You must pack this structure before you write it to the database and you must unpack this structure after you reading it from the database. ▲

```
struct OTCfgTCPInterfacesUnpacked {
    UInt16              fCount;
    UInt8               pad1;
    OTCfgTCPConfigMethodfConfigMethod;
    InetHost            fIPAddress;
    InetHost                fSubnetMask;
    Str32                   fAppleTalkZone;
    UInt8                   pad2;
    UInt8                   path[kMaxPortNameSize];
    UInt8                   module[kMaxModuleNameSize];
    UInt32                  framing;
};
```

Field descriptions

`fCount`	A value that is always 1 in the current versions of Open Transport.
`pad1`	A pad byte. Remove this pad byte when you pack this structure.
`fConfigMethod`	The configuration method. For possible values, see the section `OTCfgTCPConfigMethod` (page 162).
`fIPAddress`	The IP address that has been assigned to this interface.
`fSubnetMask`	The subnet mask.
`fAppleTalkZone`	The AppleTalk zone for this interface. Remove trailing bytes when you pack this structure.
`pad2`	A pad byte. Remove this pad byte when you pack this structure.
`path`	The name of the port over which this interface communicates.
`module`	The name of the module that controls the port over which this interface communicates.
`framing`	Ethernet framing options. Constants are defined in the file “OpenTransportProviders.h,” an OpenTransport header file.

The preference type for `OTCfgTCPInterfacesUnpacked` is `kOTCfgTCPInterfacesPref`, which is defined as `'iitf'`.

OTCfgTCPInterfacesPacked

The `OTCfgTCPInterfacesPacked` structure stores information about the configured TCP/IP interfaces in packed format. See Listing 2-16 in Chapter 2, "Using Network Setup," for sample code that packs and unpacks this structure.

IMPORTANT

You must pack this structure before you write it to the database and you must unpack this structure after you reading it from the database. ▲

```
struct OTCfgTCPInterfacesPacked {
    UInt16  fCount;
    UInt8   fConfigMethod;
    UInt8   fIPAddress[4];
    UInt8   fSubnetMask[4];
    UInt8   fAppleTalkZone[256];
    UInt8   part[sizeof(OTCfgTCPInterfacesPackedPart)];
};
```

Field descriptions

`fCount`	A value that is always 1 in the current versions of Open Transport.
`fConfigMethod`	The configuration method. For possible values, see `OTCfgTCPConfigMethod` (page 162).
`fIPAddress`	The IP address that has been assigned to this interface.
`fSubnetMask`	The subnet mask.
`fAppleTalkZone`	The AppleTalk zone for this interface. Remove trailing bytes when you pack this structure.
`part`	A `OTCfgTCPInterfacesPackedPart` (page 114) structure containing port, module, and framing information.

OTCfgTCPInterfacesPackedPart

The `OTCfgTCPInterfacesPackedPart` structure is a member of the `OTCfgTCPInterfacesPacked` (page 114) structure and stores port, module and framing information about the configured TCP/IP interfaces.

```
struct OTCfgTCPInterfacesPackedPart
{
    UInt8   path[kMaxPortNameSize];
    UInt8   module[kMaxModuleNameSize];
    UInt32  framing;
};
```

Field descriptions

`path`	The name of the port over which this interface communicates.
`module`	The name of the module that controls the port over which this interface communicates.
`framing`	Ethernet framing options. Constants are defined in the file "OpenTransportProviders.h," an Open Transport header file.

OTCfgTCPDHCPLeaseInfo

The `OTCfgTCPLeaseDHCPInfo` structure stores information about the DHCP lease for an interface.

```
struct  OTCfgTCPDHCPLeaseInfo
{
    InetHost    ipIPAddr;
    InetHost    ipConfigServer;
    UInt32      ipLeaseGrantTime;
    UInt32      ipLeaseExpirationTime;
};
```

Field descriptions

`ipIPAddr`	The IP address that has been assigned.
`ipConfigServer`	The IP address of the DHCP server.
`ipLeaseGrantTime`	The time at which the lease was acquired. The time is in seconds as returned by `GetDateTime`.
`ipLeaseExpirationTime`	The time at which the lease expires. The time is in seconds as returned by `GetDateTime`.

The preference type for `OTCfgTCPLeaseDHCPInfo` is `kOTCfgTCPDHCPLeaseInfoPref`, which is defined as `'dclt'`.

OTCfgTCPDNSServersList

The `OTCfgTCPDNSServersList` structure stores the list of name servers that have been configured for an interface.

```
struct  OTCfgTCPDNSServersList
{
    UInt16      fCount;
    InetHost    fAddressesList[1];
};
```

Field descriptions

`fCount`	The number of IP addresses in the list.
`fAddressesList`	An unbounded array containing the IP addresses of name servers.

The preference type for `OTCfgTCPDNSServersList` is `kOTCfgTCPDNSServersListPref`, which is defined as `'idns'`.

OTCfgTCPLocks

The `OTCfgTCPLocks` structure stores information about whether a preference has been locked by the administration mode of the control panel.

```
struct  OTCfgTCPLocks
{
    UInt8       pad1;
    UInt8       lockConnectViaPopup;
    UInt8       pad2;
    UInt8       lockConfigurePopup;
    UInt8       pad3;
    UInt8       lockAppleTalkZone;
    UInt8       pad4;
    UInt8       lockIPAddress;
    UInt8       pad5;
    UInt8       lockLocalDomainName;
    UInt8       pad6;
    UInt8       lockSubnetMask;
    UInt8       pad7;
    UInt8       lockRoutersList;
    UInt8       pad8;
    UInt8       lockDNSServersList;
    UInt8       pad9;
    UInt8       lockAdminDomainName;
    UInt8       pad10;
    UInt8       lockSearchDomains;
    UInt8       pad11;
    UInt8       lockUnknown;
    UInt8       pad12;
    UInt8       lock8023;
    UInt8       pad13;
    UInt8       lockDHCPClientID;
    UInt8       pad14;
};
```

Field descriptions

`pad1`	Always zero.
`lockConnectViaPopup`	Set to `TRUE` to lock the Connect Via popup menu.
`pad2`	Always zero.
`lockConfigurePopup`	Set to `TRUE` to lock the Configure popup menu.
`pad3`	Always zero.
`lockAppleTalkZone`	Set to `TRUE` to lock the AppleTalk zone that appears when the TCP/IP control panel is configured for MacIP.
`pad4`	Always zero.
`lockIPAddress`	Set to `TRUE` to lock the IP address.
`pad5`	Always zero.
`lockLocalDomainName`	Set to `TRUE` to lock the starting domain address.
`pad6`	Always zero.
`lockSubnetMask`	Set to `TRUE` to lock the “Subnet mask” text field.
`pad7`	Always zero.

`lockRoutersList`	Set to `TRUE` to lock the "Router address" text field.
`pad8`	Always zero.
`lockDNSServersList`	Set to `TRUE` to lock the "Name server addr." text field.
`pad9`	Always zero.
`lockAdminDomainName`	Set to `TRUE` to lock the "Ending domain name" text field.
`pad10`	Always zero.
`lockSearchDomains`	Set to `TRUE` to lock the "Additional search domains" text field.
`pad11`	Always zero.
`lockUnknown`	Reserved.
`pad12`	Always zero.
`lock8023`	Set to `TRUE` to lock the Use 802.3 checkbox.
`pad13`	Always zero.
`lockDHCPClientID`	Set to `TRUE` to lock the DHCP Client ID text field. This field was added to the `OTCfgTCPLocks` structure in Open Transport 2.0.
`pad14`	Always zero. This field was added to the `OTCfgTCPLocks` structure in Open Transport 2.0.

Depending on the version of Open Transport, the size of the `OTCfgTCPLocks` structure is 25 bytes (pre-Open Transport 2.0) or 27 bytes (Open Transport 2.0 and later). The following preference size constants are defined for this structure:

```
enum {
    kOTCfgTCPLocksPrefPre2_0Size  = 25,
    kOTCfgTCPLocksPref2_0Size     = 27,
    kOTCfgTCPLocksPrefCurrentSize = kOTCfgTCPLocksPref2_0Size,
};
```

When reading or writing this preference, be sure to use the appropriate preference type for the version of Open Transport that is being used.

The preference type for `OTCfgTCPLocks` is `kOTCfgTCPLocksPref`, which is defined as `'stng'`.

OTCfgTCPRoutersList

The `OTCfgTCPRoutersList` structure holds an array of `OTCfgTCPRoutersListEntry` (page 119) structures.

```
struct  OTCfgTCPRoutersList
{
    UInt16                      fCount;
    OTCfgTCPRoutersListEntry    fList[1];
};
```

Field descriptions

`fCount`	The number of elements in the `fList` array.
`fList`	An unbounded array consisting of a `OTCfgTCPRoutersListEntry` (page 119) structures.

The preference type for this preference is `kOTCfgTCPRoutersListPref`, which is defined as `'irte'`.

OTCfgTCPRoutersListEntry

The `OTCfgTCPRoutersListEntry` structure is a sub-structure of the `OTCfgTCPRoutersList` (page 119) structure and stores the IP address of the router that has been configured as the default gateway for this interface.

```
struct  OTCfgTCPRoutersListEntry
{
    InetHost    fToHost;
    InetHost    fViaHost;
    UInt16      fLocal;
    UInt16      fHost;
};
```

Field descriptions

`fToHost`	A reserved field that you should initialize to zero.
`fViaHost`	The IP address of the router.
`fLocal`	A reserved field that you should initialize to zero.
`fHost`	A reserved field that you should initialize to zero.

OTCfgTCPSearchDomains

The `OTCfgTCPSearchDomains` structure stores the list of domains that are searched after the implicit search domains.

IMPORTANT

You must pack this structure before you write it to the database and you must unpack this structure after you reading it from the database. ▲

```
struct OTCfgTCPSearchDomains {
    UInt16  fCount;
    Str255  fFirstSearchDomain;
};
typedef struct OTCfgTCPSearchDomains OTCfgTCPSearchDomains;
```

Field descriptions

`fCount`	The number of domains in the list
`fFirstSearchDomain`	The first domain to be searched. The other search domains are packed after this `fFirstSearchDomain`.

Note

This preference is stored in string list format (the same format as a ‘`STR#`’ resource). ◆

The preference type for this preference is `kOTCfgTCPSearchDomainsPref`, which is defined as `'isdm'`.

OTCfgTCPSearchList

The `OTCfgTCPSearchList` structure stores DNS configuration information.

IMPORTANT

You must pack this structure before you write it to the database and you must unpack this structure after you reading it from the database. ▲

```
struct OTCfgTCPSearchList {
    UInt8   fPrimaryInterfaceIndex;
    Str255  fLocalDomainName[256];
    Str255  fAdmindomain[256];
};
```

Field descriptions

`fPrimaryInterfaceIndex`	A value that must be 1 in the current versions of Open Transport.
`fLocalDomainName`	The local domain name in Pascal string format. You must unpack this field when you read this structure from the database and pack this file when you write this structure to the database.
`fAdmindomain`	The administrative domain name in Pascal string format. You must unpack this field when you read this structure from the database and pack this file when you write this structure to the database.

The preference type for this preference is `kOTCfgTCPSearchListPref`, which is defined as `'ihst'`.

OTCfgTCPUnloadAttr

The `OTCfgTCPUnloadAttr` enumeration defines values that indicate whether TCP/IP is loaded on demand, always loaded, or inactive. These values are used in the `kOTCfgTCPUnloadAttrPref` preference.

```
typedef UInt16 OTCfgTCPUnloadAttr
enum {
    kOTCfgTCPActiveLoadedOnDemand = 1,
    kOTCfgTCPActiveAlwaysLoaded = 2,
    kOTCfgTCPInactive = 3
};
```

Constant descriptions

`kOTCfgTCPActiveLoadedOnDemand`	TCP/IP is loaded when needed and unloaded when inactive for two minutes.

`kOTCfgTCPActiveAlwaysLoaded`
TCP/IP is always loaded.

`kOTCfgTCPInactive` TCP/IP is never loaded.

The preference type for this preference is `kOTCfgTCPDHCPUnloadAttrPref`, which is defined as `'unld'`.

Apple Remote Access Structures

This section describes structures that store Apple Remote Access (ARA) preferences. The structures are

- `OTCfgRemoteAlternateAddress` (page 123) stores an alternate number to dial.
- `OTCfgRemoteApplication` (page 123) stores information used by the Remote Access and the Open Transport/PPP applications.
- `OTCfgRemoteARAP` (page 124) stores the name of the underlying modem port.
- `OTCfgRemoteClientLocks` (page 125) stores information about whether a preference is locked.
- `OTCfgRemoteClientMisc` (page 127) stores automatic connection information.
- `OTCfgRemoteConnect` (page 127) stores core connection information for ARA configurations.
- `OTCfgRemoteDialAssist` (page 130) stores area and country code dialing information.
- `OTCfgRemoteDialing` (page 131) stores settings for outgoing ARA connections.
- `OTCfgRemoteIPCP` (page 132) stores information for configuring the Internet Protocol Control Protocol (IPCP) layer of PPP.
- `OTCfgRemoteLCP` (page 133) stores information for configuring the Link Control Protocol (LCP) layer of PPP.
- `OTCfgRemoteLogOptions` (page 135) controls the level of logging performed by ARA.
- `OTCfgRemotePassword` (page 135) holds the user's dialup password in encrypted form
- `OTCfgRemoteServer` (page 136) stores an array of port configuration IDs used to locate the configuration for a particular port on a Remote Access server.

- `OTCfgRemoteServerPort` (page 137) stores core configuration information for the personal server.
- `OTCfgRemoteTerminal` (page 138) stores information used by the PPP terminal window.
- `OTCfgRemoteUserMode` (page 139) stores the current user mode and the administration password for the control panel.
- `OTCfgRemoteX25` (page 140) stores X.25 connection information.

OTCfgRemoteAlternateAddress

The `OTCfgRemoteAlternateAddress` structure stores an alternate number to dial for outgoing ARA connections.

```
struct OTCfgRemoteAlternateAddress
{
    UInt32      pad;
    Str255      alternateAddress;
};
```

Field descriptions

`pad`	Must be zero.
`alternateAddress`	A string containing the alternate number to dial.

The preference type for this preference is `kOTCfgRemoteAlternateAddressPref`, which is defined as `'cead'`.

OTCfgRemoteApplication

The `OTCfgRemoteApplication` structure stores information for the Remote Access application (or OT/PPP).

```
struct OTCfgRemoteApplication
{
    UInt32          version;
    Point           fWindowPosition;
    UInt32          tabChoice;
    OTCfgUserMode32 fUserMode;
    UInt32          fSetupVisible;
};
```

`version`	Must be 1 for Open Transport/PPP or 3 for ARA.
`fWindowPosition`	Global coordinates for the application's window position.
`tabChoice`	Currently active tab in the Options dialog box. Use 1 for the Redialing tab, 2 for the Connection tab, or 3 for the Protocol tab.
`fUserMode`	The current user mode. See the `OTCfgUserMode` preference (page 109) enumeration for possible values.
`fSetupVisible`	Set to 1 to display the set up or zero to hide it.

The preference type for this preference is `kOTCfgRemoteApplicationPref`, which is defined as `'capt'`.

OTCfgRemoteARAP

The `OTCfgRemoteARAP` structure stores connection information used by the ARAP modules.

```
struct OTCfgRemoteARAP
{
    UInt32      version;
    char        lowerLayerName[kMaxProviderNameSize];
};
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`lowerLayerName`	A C string containing the name of the underlying modem port, which must be "Script".

The preference type for this preference is `kOTCfgRemoteARAPPref`, which is defined as `'arap'`.

OTCfgRemoteClientLocks

The `OTCfgRemoteClientLocks` structure stores information about preferences that have been locked by the administration mode of the control panel.

```
struct OTCfgRemoteClientLocks
{
    UInt32      version;
    UInt32      name;
    UInt32      password;
    UInt32      number;
    UInt32      errorCheck;
    UInt32      headerCompress;
    UInt32      termWindow;
    UInt32      reminder;
    UInt32      autoConn;
    UInt32      redial;
    UInt32      useProtocolLock;
    UInt32      useVerboseLogLock;
    UInt32      regUserOrGuestLock;
    UInt32      dialAssistLock;
    UInt32      savePasswordLock;
    UInt32      reserved[2];
};
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`name`	The Name field in the control panel is locked when the `name` field is set to 1 and unlocked when the `name` field is set to zero.
`password`	The Password field in the control panel is locked when the `password` field is set to 1 and unlocked when the `password` field is set to zero.
`number`	The Number field in the control panel is locked when the `number` field is set to 1 and unlocked when the `number` field is set to zero.

errorCheck	The "Allow error correction and compression in modem" checkbox in the control panel is locked when the errorCheck field is set to 1 and unlocked when the errorCheck field is set to zero.
headerCompress	The "Use TCP header compression" checkbox in the control panel is locked when the headerCompress field is set to 1 and unlocked when the headerCompress field is set to zero.
termWindow	The "Connect to a command-line host" checkbox in the control panel is locked when the termWindow field is set to 1 and unlocked when the termWindow field is set to zero.
reminder	The Reminders options in the control panel are locked when the reminder field is set to 1 and unlocked when the reminder field is set to zero.
autoConn	The "Connect automatically when starting TCP/IP applications" checkbox in the control panel is locked when the autoConn field is set to 1 and unlocked when the autoConn field is set to zero.
redial	The Redialing tab in the control panel is locked when the redial field is set to 1 and unlocked with the redial field is set to zero.
useProtocolLock	The "Use protocol" pop-up menu in the control panel is locked when the useProtocolLock field is set to 1 and unlocked when the useProtocolLock field is set to zero.
useVerboseLogLock	The "Use verbose logging" checkbox in the control panel is locked when the useVerboseLogLock field is set to 1 and unlocked when the useVerboseLogLock field is set to zero.
regUserOrGuestLock	The Register User and Guest radio buttons in the control panel are locked when the regUserOrGuestLock field is set to 1 and unlocked when the regUserOrGuestLock field is set to zero.
dialAssistLock	The Use DialAssist checkbox in the control panel is locked when the dialAssistLock field is set to 1 and unlocked when the dialAssistLock field is set to zero.
savePasswordLock	The "Save password" checkbox in the control panel is locked when the savePasswordLock field is set to 1 and unlocked when the savePasswordLock field is set to zero.
reserved	Must be zero.

The preference type for this preference is `OTCfgRemoteClientLocks`, which is defined as `'clks'`.

OTCfgRemoteClientMisc

The `OTCfgRemoteClientMisc` structure stores automatic connection information.

```
struct OTCfgRemoteClientMisc
{
    UInt32      version;
    UInt32      connectAutomatically;
};
```

Field descriptions

`version`
: Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.

`connectAutomatically`
: Set to 1 to connect automatically when the first TCP/IP application starts up. Set to zero to not connect automatically.

The preference type for this preference is `kOTCfgRemoteClientMiscPref`, which is defined as `'cmsc'`.

OTCfgRemoteConnect

The `OTCfgRemoteConnect` structure store core connection information for ARA configurations.

```
struct OTCfgRemoteConnect {
    UInt32                      version;
    UInt32                      fType;
    UInt32                      isGuest;
    UInt32                      canInteract;
    UInt32                      showStatus;
    UInt32                      passwordSaved;
    UInt32                      flashConnectedIcon;
    UInt32                      issueConnectedReminders;
    SInt32                      reminderMinutes;
    UInt32                      connectManually;
    UInt32                      allowModemDataCompression;
    OTCfgRemotePPPConnectScript chatMode;
    OTCfgRemoteProtocol         serialProtocolMode;
    UInt32                      passwordPtr;
    UInt32                      userNamePtr;
    UInt32                      addressLength;
    UInt32 *                    addressPtr;
    Str63                       chatScriptName;
    UInt32                      chatScriptLength;
    UInt32                      chatScriptPtr;
    UInt32                      additional;
    UInt32                      useSecurityModule;
    OSType                      securitySignature;
    UInt32                      securityDataLength;
    UInt32                      securityDataPtr;
};
typedef struct OTCfgRemoteConnect OTCfgRemoteConnect;
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`fType`	Must be zero.
`isGuest`	Set to zero if the user is a registered user; set to 1 if the user is to log on as Guest.
`canInteract`	Must be 1.
`showStatus`	Must be zero.
`passwordSaved`	Set to 1 to use the password preference (`kOTCfgRemotePasswordPref`) or set to zero to prompt the user for a password.

`flashConnectedIcon`	Set to zero if the menu bar flashes when a disconnection occurs; set to 1 if the menu bar does not flash when a disconnection occurs.
`issueConnectedReminders`	Set to 1 to flash an icon in the menu bar to remind the user that the connection is active; set to zero to not flash an icon.
`reminderMinutes`	If Notification Manager reminders are enabled, the number of minutes that are to elapse between reminders.
`connectManually`	Must be zero.
`allowModemDataCompression`	Set to zero if modem data compression is not allowed; set to 1 if modem data compression is allowed.
`chatMode`	The chat mode. For possible values, see the `OTCfgRemotePPPConnectScript` (page 166) enumeration.
`serialProtocolMode`	The serial protocol mode (PPP, ARAP, or both). For possible values, see the `OTCfgRemoteProtocol` (page 167) enumeration.
`passwordPtr`	Run-time use only. Initialize `passwordPtr` to zero, ignore it when you read it, and preserve its value when you write it.
`userNamePtr`	Run-time use only. Initialize `userNamePtr` to zero, ignore it when you read it, and preserve its value when you write it.
`addressLength`	The length in bytes of the telephone number stored in the `kOTCfgRemoteAddressPref`.
`addressPtr`	Run-time use only. Initialize `addressPtr` to zero, ignore it's value when you read it, and preserve it's value when you write it.
`chatScriptName`	The user-visible name of the chat script for this configuration.
`chatScriptLength`	The length in bytes of the chat script.
`chatScriptPtr`	Run-time use only. Initialize `chatScriptPtr` to zero, ignore it's value when you read it, and preserve it's value when you write it.
`additionalPtr`	Run-time use only. Initialize `additionalPtr` to zero, ignore it's value when you read it, and preserve it's value when you write it.
`useSecurityModule`	Must be zero.
`securitySignature`	Must be zero.

`securityDataLength`	Must be zero.
`securityData`	Must be zero.

The preference type for `OTCfgRemoteConnect` is `kOTCfgRemoteConnectPref`, which is defined as `'conn'`.

OTCfgRemoteDialAssist

The `OTCfgRemoteDialAssist` structure stores area and country code information used by the Dial Assist facility.

```
struct OTCfgRemoteDialAssist {
    UInt32  version;
    UInt32  isAssisted;
    Str31   areaCodeStr;
    Str31   countryCodeStr;
};
typedef struct OTCfgRemoteDialAssist    OTCfgRemoteDialAssist;;
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`isAssisted`	Set `isAssisted` to zero for no assistance (the default); set `isAssisted` to 1 to use Dial Assist. When `isAssisted` is set to zero, `areaCodeStr` and `countryCodeStr` are ignored.
`areaCodeStr`	A string containing an area code that is to be dialed as part of the sequence for making a connection.
`countryCodeStr`	A string containing a country code that is to be dialed as part of the sequence for making a connection.

The preference type for this preference is `kOTCfgRemoteDialAssistPref`, which is defined as `'dass'`.

OTCfgRemoteDialing

The `OTCfgRemoteDialing` structure stores settings for outgoing ARA connections.

```
struct OTCfgRemoteDialing {
    UInt32                  version;
    UInt32                  fType;
    UInt32                  additionalPtr;
    OTCfgRemoteRedialMode   dialMode;
    UInt32                  redialTries;
    UInt32                  redialDelay;
    UInt32                  pad;
};
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`fType`	Must be `‘dial’`.
`additionalPtr`	Must be zero.
`dialMode`	The redial mechanism to use. For possible values, see the `OTCfgRemoteDialMode` (page 168) enumeration.
`redialTries`	The number of times to redial if a connection cannot be made. Only valid if `dialMode` is not `kOTCfgRemoteRedialNone`.
`redialDelay`	The number of milliseconds to wait before redialing. The value of `redialDelay` is only valid if `dialMode` is not `kOTCfgRemoteRedialNone`.
`pad`	A pad byte whose value must be zero.

The preference type for this preference is `kOTCfgRemoteDialingPref`, which is defined as `'cdia'`.

OTCfgRemoteIPCP

The `OTCfgRemoteIPCP` structure stores information for configuring the Internet Protocol Control Protocol (IPCP) layer of PPP. This information is also used as part of a Remote Access server configuration. This structure is not used for ARAP connections.

```
struct OTCfgRemoteIPCP {
    UInt32      version;
    UInt32      reserved[2];
    UInt32      maxConfig;
    UInt32      maxTerminate;
    UInt32      maxFailureLocal;
    UInt32      maxFailureRemote;
    UInt32      timerPeriod;
    UInt32      localIPAddress;
    UInt32      remoteIPAddress;
    UInt32      allowAddressNegotiation;
    UInt16      idleTimerEnabled;
    UInt16      compressTCPHeaders;
    UInt32      idleTimerMilliseconds;
};
typedef struct OTCfgRemoteIPCP OTCfgRemoteIPCP;
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`reserved`	Must be zero.
`maxConfig`	Must be 10.
`maxTerminate`	Must be 10.
`maxFailureLocal`	Must be 10.
`maxFailureRemote`	Must be 10.
`timerPeriod`	In milliseconds. Must be 10000.
`localIPAddress`	Must be zero.
`remoteIPAddress`	Must be zero.

`allowAddressNegotiation`
: Must be 1.

`idleTimerEnabled`
: Set `idleTimerEnabled` to 1 to cause a connection that has been idle for the number of milliseconds specified by the `idletTimerMilliseconds` field to be disconnected. Set `idleTimerEnabled` to zero to disable the idle timer.

`compressTCPHeaders`
: Set `compressTCPHeaders` to 1 to allow Van Jacobsen header compression. Set `compressTCPHeaders` to zero to disallow header compression.

`idleTimerMilliseconds`
: The number of milliseconds to wait before disconnecting a connection that is idle.

The preference type for this preference is `kOTCfgRemoteIPCPPref`, which is defined as `'ipcp'`.

OTCfgRemoteLCP

The `OTCfgRemoteLCP` structure stores information for configuring the Link Control Protocol (LCP) layer of PPP. The information in this structure is used for PPP connections and is ignored for ARAP connections. This information is also used as part of a Remote Access server configuration.

```
struct OTCfgRemoteLCP {
    UInt32      version;
    UInt32      reserved[2];
    char        lowerLayerName[36];
    UInt32      maxConfig;
    UInt32      maxTerminate;
    UInt32      maxFailureLocal;
    UInt32      maxFailureRemote;
    UInt32      timerPeriod;
    UInt32      echoTrigger;
    UInt32      echoTimeout;
    UInt32      echoRetries;
    UInt32      compressionType;
    UInt32      mruSize;
    UInt32      upperMRULimit;
    UInt32      lowerMRULimit;
    UInt32      txACCMap;
    UInt32      rcACCMap;
    UInt32      isNoLAPB;
};
typedef struct OTCfgRemoteLCP OTCfgRemoteLCP;
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`reserved`	Must be zero.
`lowerLayerName`	A C string containing the name of the underlying modem port. Must be 'Script'.
`maxConfig`	Must be 10.
`maxTerminate`	Must be 10.
`maxFailureLocal`	Must be 10.
`maxFailureRemote`	Must be 10.
`timerPeriod`	In milliseconds. Must be 10000.
`echoTrigger`	In milliseconds. Must be 10000.
`echoTimeout`	In milliseconds. Must be 10000.
`echoRetries`	Must be 5.
`compressionType`	Must be 3.
`mruSize`	Must be 1500.
`upperMRULimit`	Must be 4500.
`lowerMRULimit`	Must be zero.
`txACCMap`	Must be zero.
`rcACCMap`	Must be zero.
`isNoLAPB`	Must be zero.

The preference type for this preference is `kOTCfgRemoteLCPPref`, which is defined as `'lcp'`.

OTCfgRemoteLogOptions

The `OTCfgRemoteLogOptions` structure controls the level of logging performed by ARA.

```
struct OTCfgRemoteLogOptions {
    UInt32  version;
    UInt32  fType;
    UInt32  additionalPtr;
    OTCfgRemoteLogLevel logLevel;
    UInt32  reserved[4];
};
typedef struct OTCfgRemoteLogOptions OTCfgRemoteLogOptions;
```

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`fType`	Must be `‘lgop’`.
`additional`	Run-time use only. Initialize to zero. When reading, ignore the value of `additionalPtr`. When writing `additionalPtr`, preserve its value.
`logLevel`	The log level. For possible values, see the `OTCfgRemoteLogLevel` (page 168) enumeration.
`reserved`	Reserved.

The preference type for this preference is `kOTCfgRemoteLogOptionsPref`, which is defined as `'logo'`.

OTCfgRemotePassword

The `OTCfgRemotePassword` structure holds the user’s dialup password in encrypted form. For sample code, see Listing 2-17 in Chapter 2, “Using Network Setup.”

```
struct OTCfgRemotePassword {
    UInt8   data[256];
};
typedef struct OTCfgRemotePassword OTCfgRemotePassword;
```

`data`	The encrypted password. Call `OTCfgEncrypt` (page 93) to encrypt the password.

The preference type for this preference is `kOTCfgRemotePasswordPref`, which is defined as `'pass'`.

OTCfgRemoteServer

The `OTCfgRemoteServer` structure stores an array of port configuration IDs used to locate the configuration for a particular port.

```
struct OTCfgRemoteServer {
    UInt32  version;
    SInt16  configCount;
    SInt16  configIDs[1];
};
```

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`configCount`	The number of active Remote Access server configurations. Must be 1 for the personal server.
`configIDs`	Array of port configuration IDs. For the personal server, there can be only one port configuration ID whose value is zero.

The preference type for this preference is `kOTCfgRemoteServerPref`, which is defined as `'srvr'`.

OTCfgRemoteServerPort

The `OTCfgRemoteServerPort` structure stores core configuration information for the personal server.

```
struct OTCfgRemoteServerPort {
    UInt32                      version;
    SInt16                      configID;
    Str255                      password;
    OTCfgRemoteAnswerMode       answerMode;
    Boolean                     limitConnectTime;
    UInt8                       pad;
    UInt32                      maxConnectSeconds;
    OTCfgRemoteProtocol         serialProtoFlags;
    OTCfgRemoteNetworkProtocol  networkProtoFlags;
    OTCfgRemoteNetAccessMode    netAccessMode;
    Boolean                     requiresCCL;
    char                        portName[64];
    char                        serialLayerName[kMaxProviderNameSize];
    InetHost                    localIPAddress;
};
```

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`configID`	The ID of this port configuration. The ID must match an element of the `configIDs` array in the `OTCfgRemoteServer` (page 136) structure. For the personal server, `configID` must be zero.
`password`	The security zone bypass password in plain text.
`answerMode`	The answer mode. For possible values, see the `OTCfgRemoteAnswerMode` (page 169) enumerations.
`limitConnectTime`	Set to 1 to limit the length of incoming connections. Set to zero for unlimited connection time.
`pad`	Must be zero.
`maxConnectSeconds`	The maximum length of a incoming connection in seconds if `limitConnectTime` is set to 1. The default is 3600.

`serialProtoFlags`	Serial protocol flags. For possible values, see the `OTCfgRemoteProtocol` (page 167).
`networkProtoFlags`	Network protocol flags. For possible values, see the `OTCfgRemoteNetworkProtocol` (page 169) enumeration.
`netAccessMode`	Access mode flags. For possible values, see the `OTCfgRemoteNetAccessMode` (page 170) enumeration.
`requiresCCL`	Must be `TRUE`.
`portName`	C string containing the name of the underlying port.Must the empty string for the personal server.
`serialLayerName`	C string containing the Open Transport name of the serial port.
`localIPAddress`	IP address to offer to the client.

The preference type for `OTCfgRemoteServerPort` is `kOTCfgRemoteServerPortPref`, which is defined as `'port'`.

OTCfgRemoteTerminal

The `OTCfgRemoteTerminal` structure stores information used by the PPP terminal window.

```
struct OTCfgRemoteTerminal {
    UInt32      fVersion;
    Boolean     fLocalEcho;
    Boolean     fNonModal;
    Boolean     fPowerUser;
    Boolean     fQuitWhenPPPStarts;
    Boolean     fDontAskVarStr;
    Boolean     fNoVarStrReplace;
    Boolean     fLFAfterCR;
    Boolean     fAskToSaveOnQuit;
    Rect        fWindowRect;
    Style       fTypedCharStyle;
    Style       fPrintedCharStyle;
    Style       fEchoedCharStyle;
    UInt8       pad;
    SInt16      fFontSize;
    Str255      fFontName;
};
typedef struct OTCfgRemoteTerminal OTCfgRemoteTerminal;
```

`fVersion`	Must be 1.
`fLocalEcho`	Set to `TRUE` for the terminal window to echo typed characters; otherwise, set to `FALSE`. The default is `FALSE`.
`fNonModal`	Must be `FALSE`.
`fPowerUser`	Must be `FALSE`.
`fQuitWhenPPPStarts`	Set to `TRUE` to cause the terminal window to quit when the PPP connection is made. The default is `TRUE`.
`fDontAskVarStr`	The default is `FALSE`.
`fNoVarStrReplace`	Must be `FALSE`.
`fLFAfterCR`	Must be `FALSE`.
`fAskToSaveOnQuit`	Set to `TRUE` to cause ARA to ask to save changes when the terminal window closes. The default is `FALSE`.
`fWindowRect`	Must be zero.
`fTypedCharStyle`	Style used for typed characters. The default is bold.
`fPrintedCharStyle`	Style used for characters sent by the other end of the connection. The default is plain.
`fEchoedCharStyle`	Style used for echoed characters. The default is italic.
`pad`	Must be zero.
`fFontSize`	The font size. The default is 9 point.
`fFontName`	The font in which characters are displayed. The default is Monaco on Roman systems.

The preference type for this preference is `kOTCfgRemoteTerminalPref`, which is defined as `'term'`.

OTCfgRemoteUserMode

The `OTCfgRemoteUserMode` structure stores the current user mode and the administration password.

```
struct OTCfgRemoteUserMode {
    UInt32          version;
    OTCfgUserMode32 userMode;
    Str255          adminPassword;
};
typedef struct OTCfgRemoteUserMode OTCfgRemoteUserMode;
```

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`userMode`	Current user mode. See the `OTCfgUserMode` preference (page 109) enumeration for possible values.
`adminPassword`	The administration password. The format is not documented.

The preference type for this preference is `kOTCfgRemoteUserModePref`, which is defined as `'usmd'`.

OTCfgRemoteX25

The `OTCfgRemoteX25` structure stores X.25 connection information.

```
struct OTCfgRemoteX25 {
    UInt32      version;
    UInt32      fType;
    UInt32      additionalPtr;
    FSSpec      script;
    UInt8       address[256];
    UInt8       userName[256];
    UInt8       closedUserGroup[5];
    Boolean     reverseCharge;
};
```

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version`

	field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`fType`	Must be zero for standard dial-up connections.
`additionalPtr`	Must be zero for standard dial-up connections.
`script`	Must be zero for standard dial-up connections.
`address`	Must be zero for standard dial-up connections.
`userName`	Must be zero for standard dial-up connections.
`closedUserGroup`	Must be zero for standard dial-up connections.
`reverseCharge`	Must be zero for standard dial-up connections.

The preference type for this preference is `kOTCfgRemoteX25Pref`, which is defined as `'x25 '`.

IMPORTANT

Using Network Setup to configure X.25 connections is not supported. ▲

Modem Structures

This section describes structures that store Modem control panel settings. The structures are

- `OTCfgModemGeneral` (page 141) stores per-connection modem preferences.
- `OTCfgModemApplication` (page 142) stores the current user mode setting and the window position of the Modem control panel.
- `OTCfgModemLocks` (page 143) stores the lock settings for the Modem control panel.

OTCfgModemGeneral

The `OTCfgModemGeneral` structure stores most of the per-connection modem preferences.

```
struct  OTCfgModemGeneral
{
    UInt32                      version;
    Boolean                     useModemScript;
    UInt8                       pad;
    FSSpec                      modemScript;
    Boolean                     modemSpeakerOn;
    Boolean                     modemPulseDial;
    OTCfgModemDialogToneMode    modemDialToneMode;
    char                        lowerLayerName[kMaxProviderNameSize];
};
```

Field descriptions

`version`	Depending how the preference was constructed, `version` may be `kOTCfgRemoteDefaultVersion` or `kOTCfgRemoteAcceptedVersion`. When reading the `version` field, accept either value. When writing the `version` field, set it to `kOTCfgRemoteDefaultVersion`.
`useModemScript`	Set `useModemScript` to `TRUE` to indicate that a modem script is to be used.
`pad`	A value that must be zero.
`modemScript`	The modem script that is to be used; this field is ignored if a modem script is not to be used.
`modemSpeakerOn`	Set `modemSpeakerOn` to `TRUE` to indicate dialing with the modem speaker on. Otherwise, set `modemSpeakerOn` to `FALSE`.
`modemPulseDial`	Set `modemPulseDial` to `TRUE` to indicate pulse dialing. Otherwise, set `modemPulseDial` to `FALSE` for tone dialing.
`modemDialToneMode`	The dial tone mode that controls the way in which the modem handles dial tone. For possible values, see the `OTCfgModemDialingToneMode` (page 171) enumeration.
`lowerLayerName`	The name of the underlying serial port in C string format.

The preference type for the `OTCfgModemGeneral` structure is `kOTCfgModemGeneralPrefs`, which is defined as `'ccl '`.

OTCfgModemApplication

The `OTCfgModemApplication` structure stores the current user mode setting and the window position of the Modem control panel.

```
struct OTCfgModemApplication {
    UInt32          version;
    Point           windowPos;
    OTCfgUserMode32 userMode;
};
```

Field descriptions

`version`	Must be 1.
`windowPos`	Window position in global coordinates of the modem control panel.
`userMode`	Must be `kOTCfgBasicUserModeUser` mode because the Modem control panel does not support any other mode.

The preference type for the `OTCfgModemApplication` structure is `kOTCfgModemApplicationPref`, which is defined as `'mapt'`.

OTCfgModemLocks

The `OTCfgModemLocks` structure stores the lock settings for the Modem control panel.

```
struct OTCfgModemLocks {
    UInt32  version;
    UInt32  port;
    UInt32  script;
    UInt32  speaker;
    UInt32  dialing;
};
typedef struct OTCfgModemLocks OTCfgModemLocks;
```

Field descriptions

`version`	Must be 1.
`port`	Set `port` to 1 to lock the setting for the underlying serial port or to zero to unlock the setting.
`script`	Set script to 1 to lock the modem script (CCL) or to zero to unlock the setting.
`speaker`	Set `speaker` to 1 to lock the speaker setting or to zero to unlock the speaker setting.

`dialing`	Set `dialing` to lock the setting for pulse or tone dialing, or set `dialing` to zero to unlock the setting.

The preference type for the `OTCfgModemLocks` structure is `kOTCfgModemLocksPref`, which is defined as `'lkmd'`.

AppleTalk Structures

This section describes the structures that store AppleTalk preferences. The structures are

- `OTCfgATalkGeneral` (page 145) is a general structure holds the combined preferences for each AppleTalk protocol.
- `OTCfgATalkGeneralAARP` (page 146) stores information for the AppleTalk Address Resolution protocol (AARP).
- `OTCfgATalkGeneralADSP` (page 147) stores information for the AppleTalk Data Stream Protocol (ADSP).
- `OTCfgATalkGeneralASP` (page 149) stores information for the AppleTalk Session Protocol (ASP).
- `OTCfgATalkGeneralATP` (page 150) stores information for the AppleTalk Transaction Protocol (ATP).
- `OTCfgATalkGeneralDDP` (page 151) stores information for the Datagram Delivery Protocol (DDP).
- `OTCfgATalkGeneralNBP` (page 153) stores information for the Network Binding Protocol (NBP).
- `OTCfgATalkGeneralPAP` (page 154) stores information for the Printer Access Protocol (PAP).
- `OTCfgATalkGeneralZIP` (page 155) stores information for the Zone Information Protocol (ZIP).
- `OTCfgATalkLocks` (page 156) stores information about whether AppleTalk preferences have been locked.
- `OTCfgATalkNetworkArchitecture` (page 157) stores information about whether classic networking or Open Transport is selected.
- `OTCfgATalkPortDeviceType` (page 158) stores information about the port for which AppleTalk is configured.

OTCfgATalkGeneral

The `OTCfgATalkGeneral` structure consists of structures for each AppleTalk protocol.

```
struct  OTCfgATalkGeneral
{
    UInt16                  fVersion;
    UInt16                  fNumPrefs;
    OTPortRef               fPort;
    void*                   fLink;
    void*                   fPrefs[8];
    OTCfgATalkGeneralAARP   aarpPrefs;
    OTCfgATalkGeneralDDP    ddpPrefs;
    OTCfgATalkGeneralNBP    nbpPrefs;
    OTCfgATalkGeneralZIP    zipPrefs;
    OTCfgATalkGeneralATP    atpPrefs;
    OTCfgATalkGeneralADSP   adspPrefs;
    OTCfgATalkGeneralPAP    papPrefs;
    OTCfgATalkGeneralASP    aspPrefs;
};
```

Field descriptions

`fVersion`	Must be zero.
`fNumPrefs`	Must be zero.
`OTPortRef`	A reference to the port to which this configuration applies.
`fLink`	Must be null. This field is used during run time.
`fPrefs`	All elements must be initialized to null. This field is used during run time.
`aarpPrefs`	An `OTCfgATalkGeneralAARP` (page 146) structure.
`ddpPrefs`	An `OTCfgATalkGeneralDDP` (page 151) structure.
`nbpPrefs`	An `OTCfgATalkGeneralNBP` (page 153) structure.
`zipPrefs`	An `OTCfgATalkGeneralZIP` (page 155) structure.
`adspPrefs`	An `OTCfgATalkGeneralADSP` (page 147) structure.
`papPrefs`	An `OTCfgATalkGeneralPAP` (page 154) structure.
`aspPrefs`	An `OTCfgATalkGeneralASP` (page 149) structure.

The preference type for the `OTCfgATalkGeneral` structure is `kOTCfgATalkGeneralPref`, which is defined as `'atpf'`.

OTCfgATalkGeneralAARP

The `OTCfgATalkGeneralAARP` structure defines parameters for the AppleTalk Address Resolution Protocol (AARP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct OTCfgATalkGeneralAARP {
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fAgingCount;
    UInt32      fAgingInterval;
    UInt32      fProtAddrLen;
    UInt32      fHWAddrLen;
    UInt32      fMaxEntries;
    UInt32      fProbeInterval;
    UInt32      fProbeRetryCount;
    UInt32      fRequestInterval;
    UInt32      fRequestRetryCount;
};
typedef struct OTCfgAARPPrefs OTCfgAARPPrefs;
```

Field descriptions

`fVersion`	Always 1.
`fSize`	The size of this structure.
`fAgingCount`	The default is 8.
`fAgingInterval`	The aging interval in milliseconds. The default is 1000.
`fProtAddrLen`	The length of protocol addresses in bytes. Always 4. This field is ignored by current versions of Open Transport.
`fHWAddrLen`	The length of hardware addresses. Always 6. This field is ignored by current versions of Open Transport.
`fMaxEntries`	The default is 100.
`fProbeInterval`	The probe interval in milliseconds. The default probe interval is 200 milliseconds.

`fProbeRetryCount`	The number of times to retry a probe. The default is 10.
`fRequestInterval`	The request interval in milliseconds. The default request interval is 200 milliseconds.
`fRequestRetryCount`	The number of times to retry a request. The default is 8.

For a detailed descriptions of AARP, see *Inside AppleTalk,* Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralADSP

The `OTCfgATalkGeneralADSP` structure defines parameters for the AppleTalk Data Stream Protocol (ADSP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct  OTCfgATalkGeneralADSP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fDefaultSendBlocking;
    UInt32      fTSDUSize;
    UInt32      fETSDUSize;
    UInt32      fDefaultOpenInterval;
    UInt32      fDefaultProbeInterval;
    UInt32      fMinRoundTripTime;
    UInt32      fDefaultSendInterval;
    UInt32      fDefaultRecvWindow;
    UInt8       fDefaultOpenRetries;
    UInt8       fDefaultBadSeqMax;
    UInt8       fDefaultProbeRetries;
    UInt8       fMaxConsecutiveDataPackets;
    Boolean     fDefaultChecksum;
    Boolean     fDefaultEOM;
};
```

Field descriptions

`fVersion`	Must be 1.

`fSize`	Must be the size in bytes of this structure.
`fDefaultSendBlocking`	Bytes, default is 16.
`fTSDUSize`	The Transport Service Data Unit (TSDU), which is the maximum amount of data that packets of this protocol can carry. The default is 572.
`fETSDUSize`	The extended TSDU (ETSDU) size. The default is 572.
`fDefaultOpenInterval`	The default open interval in milliseconds. The default is 3000.
`fDefaultProbeInterval`	The default probe interval in milliseconds. The default is 30000.
`fMinRoundTripTime`	The minimum round trip time in milliseconds. The default is 100.
`fDefaultSendInterval`	The default send interval in milliseconds. The default is 100.
`fDefaultRecvWindow`	The default receive window in bytes. Must be 27648. This value is ignored by current versions of Open Transport.
`fDefaultOpenRetries`	The default number of open retries allowed. The default value is 3.
`fDefaultBadSeqMax`	The default maximum number of sequence errors. The default value is 3.
`fDefaultProbeRetries`	The default number of probe retries. The default value is 3.
`fMaxConsecutiveDataPackets`	The maximum number of consecutive data packets. The default value is 48.
`fDefaultChecksum`	Whether checksumming is enabled. The default value is `FALSE`.
`fDefaultEOM`	The default end of header. The default value is `FALSE`.

For a detailed description ADSP, see *Inside Macintosh: Networking with Open Transport* and *Inside AppleTalk,* Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralASP

The `OTCfgATalkGeneralASP` structure defines parameters for the AppleTalk Session Protocol (ASP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

IMPORTANT

Open Transport does not currently include a native implementation of ASP. The classic AppleTalk implementation of ASP ignores these preferences. ▲

```
struct  OTCfgATalkGeneralASP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fDefaultTickleInterval;
    UInt8       fDefaultTickleRetries;
    UInt8       fDefaultReplies;
};
```

Field descriptions

`fVersion`	Must be 1.
`fSize`	The size in bytes of this structure.
`fDefaultTickleInterval`	The default tickle interval in milliseconds. This value must be 30000. This value is ignored by current versions of Open Transport.
`fDefaultTickleRetries`	The default number of times to retry sending a tickle. The default value is 8. This value is ignored by current versions of Open Transport.
`fDefaultReplies`	Must be 8. This field is ignored by current versions of Open Transport.

For a detailed description of ASP, see *Inside AppleTalk,* Second Edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralATP

The `OTCfgATalkGeneralATP` structure defines parameters for the AppleTalk Transaction Protocol (ATP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct  OTCfgATalkGeneralATP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fTSDUSize;
    UInt32      fDefaultRetryInterval;
    UInt32      fDefaultRetryCount;
    UInt8       fDefaultReleaseTimer;
    Boolean     fDefaultALOSetting;
};
```

Field descriptions

`fVersion`	Must be 1.
`fSize`	Must be `sizeof(OTCfgATalkGeneralATP)`.
`fTSDUSize`	The maximum amount of data that packets of this protocol can carry. The default is 578.
`fDefaultRetryInterval`	The default retry interval in milliseconds. By default, this value is 2000.
`fDefaultRetryCount`	The default retry count. By default, this value is 8.
`fDefaultReleaseTimer`	The default release timer. The default value is zero. This field has the same format as `ATP_OPT_RELTIMER` which is described in *Inside Macintosh: Networking with Open Transport.*

`fDefaultALOSetting`	The default "at least once" (ALO) setting. The default value is `FALSE`.

For a detailed descriptions ATP, see *Inside Macintosh: Networking with Open Transport* and *Inside AppleTalk*, Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralDDP

The `OTCfgATalkGeneralDDP` structure defines parameters for the Datagram Delivery Protocol (DDP) component of the AppleTalk protocol stack.

```
struct OTCfgATalkGeneralDDP {
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fTSDUSize;
    UInt8       fLoadType;
    UInt8       fNode;
    UInt16      fNetwork;
    UInt16      fRTMPRequestLimit;
    UInt16      fRTMPRequestInterval;
    UInt32      fAddressGenLimit;
    UInt32      fBRCAgingInterval;
    UInt32      fRTMPAgingInterval;
    UInt32      fMaxAddrTries;
    Boolean     fDefaultChecksum;
    Boolean     fIsFixedNode;
    UInt8       fMyZone[kZIPMaxZoneLength+1];
};
typedef struct OTCfgATalkGeneralDDP OTCfgATalkGeneralDDP;
```

Field descriptions

`fVersion`	Must be 1.
`fSize`	The size of this structure.

`fTSDUSize`	The maximum amount of data that packets of this protocol can carry. Must be 586, which is the basic AppleTalk datagram size.
`fLoadType`	Whether AppleTalk is active. See discussion below for possible values.
`fNode`	Most recently acquired node number or the fixed node number to use.
`fNetwork`	Most recently acquired network number or the fixed network number.
`fRTMPRequestLimit`	Must be 3. This field is ignored by current versions of Open Transport.
`fRTMPRequestInterval`	The request interval in milliseconds. This field must be 200. This field is ignored by current versions of Open Transport.
`fAddressGenLimit`	Address generation limit. The default is 250.
`fBRCAgingInterval`	The Best Routing Cache (BRC) aging interval in milliseconds. This field must be 4000. This field is ignored by current versions of Open Transport.
`fRTMPAgingInterval`	The Router Table Maintenance Protocol (RTMP) aging interval in milliseconds. This field must be 5000. This field is ignored by current versions of Open Transport.
`fMaxAddrTries`	The maximum number of retries that OT makes when trying to acquire an address. The default is 4096.
`fDefaultChecksum`	When set to `TRUE`, a checksum is performed on the DDP packet. When set to `FALSE`, no checksum is performed. The default is `FALSE`.
`fIsFixedNode`	Set to `TRUE` when fixed node and network numbers are being used. The default value is `FALSE`.
`fMyZone`	The most recently acquired zone.

For a detailed description of DDP, see *Inside Macintosh: Networking with Open Transport* and *Inside AppleTalk,* Second edition.

The value of the `fLoadType` field controls whether AppleTalk is active. The original definition of this field was as an inactivity timeout (in minutes), similar to the inactivity timeout implemented for TCP/IP in current versions of Open Transport.

Before Open Transport 1.0 was released, it was realized that loading and unloading AppleTalk on demand was not possible, so the `fLoadType` field was redefined as a flag, with zero meaning inactive and non-zero meaning active. However, the default preferences were not updated to reflect this change. So, it is possible to see the following values stored in this field:

```
typedef UInt8 OTCfgATalkUnloadOptions
enum {
    kOTCfgATalkInactive            = 0,
    kOTCfgATalkDefaultUnloadTimeout = 5,
    kOTCfgATalkActive              = 0xFF
};
```

When reading, treat a value of zero as meaning that AppleTalk is inactive, and treat any non-zero values as meaning that AppleTalk is active. When writing, set `fLoadType` to `kOTCfgATalkInactive` or `kOTCfgATalkActive`. Never set `fLoadType` to `kOTCfgATalkDefaultUnloadTimeout`.

No preference type is defined for the `OTCfgATalkGeneralDDP` structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralNBP

The `OTCfgATalkGeneralNBP` structure defines parameters for the Name Bind Protocol (NBP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct  OTCfgATalkGeneralNBP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fTSDUSize;
    UInt32      fDefaultRetryInterval;
    UInt32      fDefaultRetryCount;
    Boolean     fCaseSensitiveCompare;
    UInt8       fPad;
};
```

Field descriptions

`fVersion`	Must be 1.

`fSize`	The size in bytes of this structure.
`fTSDUSize`	The maximum amount of data that packets of this protocol can carry. The default is 584.
`fDefaultRetryInterval`	The default retry interval in milliseconds. By default, this value is 800.
`fDefaultRetryCount`	The default retry count. By default, this value is 3.
`fCaseSensitiveCompare`	Whether comparisons are case sensitive. The default value is `FALSE`.
`fPad`	A pad byte whose value must be zero.

For a detailed description of NBP, see *Inside Macintosh: Networking with Open Transport* and *Inside AppleTalk,* Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralPAP

The `OTCfgATalkGeneralPAP` structure defines parameters for the Printer Access Protocol (PAP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct  OTCfgATalkGeneralPAP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fDefaultOpenInterval;
    UInt32      fDefaultTickleInterval;
    UInt8       fDefaultOpenRetries;
    UInt8       fDefaultTickleRetries;
    UInt8       fDefaultReplies;
    Boolean     fDefaultPAPEOMEnabled;
};
```

Field descriptions

`fVersion`	Must be 1.
`fSize`	The size in bytes of this structure.
`fDefaultOpenInterval`	The default open interval in milliseconds. The default value is 2000.
`fDefaultTickleInterval`	The default tickle interval in milliseconds. The default value is 15000.
`fDefaultOpenRetries`	The default number of times to retry an opening. The default value is 0.
`fDefaultTickleRetries`	The default number of times to retry sending a tickle. The default value is 8.
`fDefaultReplies`	Must be 8. This field is ignored by current versions of Open Transport.
`fDefaultPAPEOMEnabled`	By default, `FALSE`.

For a detailed description of PAP, see *Inside Macintosh: Networking with Open Transport* and *Inside AppleTalk*, Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkGeneralZIP

The `OTCfgATalkGeneralZIP` structure defines parameters for the Zone Information Protocol (ZIP) component of the AppleTalk protocol stack and is a sub-structure of the `OTCfgATalkGeneral` structure.

```
struct  OTCfgATalkGeneralZIP
{
    UInt16      fVersion;
    UInt16      fSize;
    UInt32      fGetZoneInterval;
    UInt32      fZoneListInterval;
    UInt16      fDDPInfoTimeout;
    UInt8       fGetZoneRetries;
    UInt8       fZoneListRetries;
    Boolean     fChecksumFlag;
    UInt8       fPad;
};
```

Field descriptions

`fVersion`	Must be 1.
`fSize`	The size in bytes of this structure.
`fGetZoneInterval`	The "get zone" interval in milliseconds. The default is 2000.
`fZoneListInterval`	The "zone list" interval in milliseconds. The default is 2000.
`fDDPInfoTimeout`	The "DDP info" timeout in milliseconds. The default is 4000.
`fGetZoneRetries`	The "get zone" retry limit. The default is 4.
`fZoneListRetries`	The "zone list" retry limit. The default is 4.
`fChecksumFlag`	Whether checksumming is enabled. The default is zero.
`fPad`	A pad byte whose value must be zero.

For a detailed description of ZIP, see *Inside AppleTalk,* Second edition.

No preference type is defined for this structure. Instead, access this structure through the `OTCfgATalkGeneral` (page 145) structure, which has a preference type of `kOTCfgATalkGeneralPref`.

OTCfgATalkLocks

The `OTCfgATalkLocks` structure stores information about whether AppleTalk preferences have been locked by the administration mode in the control panel.

```
struct OTCfgATalkLocks
{
    UInt16 fLocks;
};
```

Field descriptions

`fLocks`	A bit field.

The following enumeration defines mask values for the `fLocks` field:

```
enum {
    kOTCfgATalkPortLockMask      =   0x01,
    kOTCfgATalkZoneLockMask      =   0x02,
    kOTCfgATalkAddressLockMask   =   0x04,
    kOTCfgATalkConnectionLockMask=   0x08,
    kOTCfgATalkSharingLockMask   =   0x10
};
```

Constant descriptions

`kOTCfgATalkPortLockMask`
The bit set by this mask indicates that the port used by AppleTalk is locked.

`kOTCfgATalkZoneLockMask`
The bit set by this mask indicates that the AppleTalk zone is locked.

`kOTCfgATalkAddressLockMask`
The bit set by this mask indicates that the AppleTalk address is locked.

`kOTCfgATalkConnectionLockMask`
The bit set by this mask indicates that the AppleTalk Connection pop-up menu is locked.

`kOTCfgATalkSharingLockMask`
Reserved.

The preference type for the `OTCfgATalkLocks` structure is `kOTCfgATalkLocksPref`, which is defined as `'lcks'`.

OTCfgATalkNetworkArchitecture

The `OTCfgATalkNetworkArchitecture` structure was used by the Network Software Selector in System 7.5.3 through 7.5.5. Despite its name and location, this preference controlled both AppleTalk and TCP/IP services.

```
struct OTCfgATalkNetworkArchitecture
{
    UInt32  fVersion;
    OSType  fNetworkArchitecture;
};
typedef struct OTCfgATalkNetworkArchitecture OTCfgATalkNetworkArchitecture;
```

Field descriptions

`fVersion`	Must be zero.
`fNetworkArchitecture`	Must be `'OTOn'`.

The preference type for the `OTCfgATalkNetworkArchitecture` structure is `kOTCfgATalkNetworkArchitecturePref`, which is defined as `'neta'`.

OTCfgATalkPortDeviceType

The `OTCfgATalkPortDeviceType` structure stores information about the port for which AppleTalk is configured. This structure is not used by the AppleTalk protocol stack, but it is used by the current AppleTalk control panel.

```
struct OTCfgATalkPortDeviceType
{
    UInt16  fDeviceType;
};
```

Field descriptions

`fDeviceType`	The Open Transport device type (such as `kOTEthernetDevice`) or an ADEV ID for the current port.

The preference type for the `OTCfgATalkPortDeviceType` structure is `kOTCfgATalkPortDeviceTypePref`, which is defined as `'ptfm'`.

Infrared Structures

This section describes structure that stores Infrared preferences. The structure is

- `OTCfgIRGeneral` (page 159)

OTCfgIRGeneral

The `OTCfgIRGeneral` structure stores per-connection infrared settings.

```
struct OTCfgIRGeneral
{
    UInt32          fVersion;
    OTPortRef       fPortRef;
    OTCfgIRPortSettingfPortSetting;
    Boolean         fNotifyOnDisconnect;
    Boolean         fDisplayIRControlStrip;
};
```

Field descriptions

`fVersion`	Must be zero.
`fPortRef`	Reference to the infrared port.
`OTCfgIRPortSetting`	A value that specifies the infrared protocol. For possible values, see the `OTCfgIRPortSetting` (page 174) enumeration.
`fNotifyOnDisconnect`	`TRUE` if the user is to be notified when the IrDA protocol disconnects; otherwise, `FALSE`.
`fDisplayIRControlStrip`	`TRUE` if the Infrared control strip is to be displayed; otherwise, `FALSE`.

The preference type for the `OTCfgIRGeneral` structure is `kOTCfgIRGeneralPref`, which is defined as `'atpf'`.

Protocol Constants and Other Data Types

The following sections describe constants and other data types that are defined for the protocols that use Network Setup:

- "TCP/IP Constants and Other Data Types" (page 160)
- "Apple Remote Access Constants and Other Data Types" (page 163)
- "Modem Constants and Other Data Types" (page 170)

- “AppleTalk Constants and Other Data Types” (page 172)
- “Infrared Constants and Other Data Types” (page 173)

TCP/IP Constants and Other Data Types

The following enumeration defines type codes for the TCP/IP preferences.

```
enum {
    kOTCfgTCPInterfacesPref     = 'iitf',
    kOTCfgTCPDeviceTypePref     = 'dtyp',
    kOTCfgTCPRoutersListPref    = 'irte',
    kOTCfgTCPSearchListPref     = 'ihst',
    kOTCfgTCPDNSServersListPref = 'idns',
    kOTCfgTCPSearchDomainsPref  = 'isdm',
    kOTCfgTCPDHCPLeaseInfoPref  = 'dclt',
    kOTCfgTCPDHCPClientIDPref   = 'dcid',
    kOTCfgTCPUnloadAttrPref     = 'unld',
    kOTCfgTCPLocksPref          = 'stng',
    kOTCfgTCPPushBelowIPPref    = 'crpt',
    kOTCfgTCPPushBelowIPListPref= 'blip',
};
```

Constant descriptions

`kOTCfgTCPInterfacesPref`
Preference type for the `OTCfgTCPInterfacesUnpacked` (page 112) structure.

`kOTCfgTCPDeviceTypePref`
Preference type for the TCP device type. Constants are defined in *Inside Macintosh: Networking with Open Transport* available at `http://developer.apple.com/techpubs/mac/NetworkingOT/NetworkingWOT-2.html`.

`kOTCfgTCPRoutersListPref`
Preference type for the `OTCfgTCPRoutersList` (page 119) structure.

`kOTCfgTCPSearchListPref`
Preference type for the `OTCfgTCPSearchList` (page 120) structure.

`kOTCfgTCPDNSServersListPref`
: Preference type for the `OTCfgTCPDNSServersList` (page 116) structure.

`kOTCfgTCPSearchDomainsPref`
: Preference type for the `OTCfgTCPSearchDomains` (page 120) structure.

`kOTCfgTCPDNSServersListPref`
: Preference type for the `OTCfgTCPDNSServersList` (page 116) structure.

`kOTCfgTCPDHCPLeaseInfoPref`
: Preference type for the `OTCfgTCPDHCPLeaseInfo` (page 115) structure.

`kOTCfgTCPDHCPClientIDPref`
: Preference type for the DHCP client ID, stored as a Pascal string.

`kOTCfgTCPUnloadAttrPref`
: Preference type for the `OTCfgTCPUnloadAttr` (page 121) structure.

`kOTCfgTCPLocksPref` Preference type for the `OTCfgTCPLocks` (page 116) structure.

`kOTCfgTCPPushBelowIPPref`
: Preference type for a Pascal string containing the name of a module to be pushed below IP.

`kOTCfgTCPPushBelowIPListPref`
: Preference type for a list of modules to be pushed below IP in `'STR#'` resource format.

Masks for the kOTCfgProtocolOptionsPref Preference

The following enumeration defines masks for the `kOTCfgProtocolOptionsPref` preference when it is in a TCP/IP entity:

```
enum {
    kDontDoPMTUDiscoveryMask        = 0x0001,
    kDontShutDownOnARPCollisionMask = 0x0002,
    kDHCPInformMask                 = 0x0004,
    kOversizeOffNetPacketsMask      = 0x0008,
    kDHCPDontPreserveLeaseMask      = 0x0010,
};
```

Constant descriptions

`kDontDoPMTUDiscoveryMask`	If set, this bit turns off path MTU discovery.
`kDontShutDownOnARPCollisionMask`	If set, this bit disables ARP collision shutdown.
`kDHCPInformMask`	If set, this bit enables DHCPINFORM instead of DHCPREQUEST.
`kOversizeOffNetPacketsMask`	If set and with path MTU discovery off, this bit disables limiting off-network packets to 576 bytes.
`kDHCPDontPreserveLeaseMask`	If set, this bit disables DHCP INIT-REBOOT capability.

For details about `kDHCPInformMask` and `kDHCPDontPreserveLeaseMask`, see Tech Info Library article 58372 available at http://til.info.apple.com/techinfo.nsf/artnum/n58372.

OTCfgTCPConfigMethod

The `OTCfgTCPConfigMethod` enumeration defines values that indicate how the interface acquires an IP address:

```
enum UInt8 OTCfgTCPConfigMethod {
    kOTCfgManualConfig,
    kOTCfgRARPConfig,
    kOTCfgBOOTPConfig,
    kOTCfgDHCPConfig,
    kOTCfgMacIPConfig
};
```

Constant descriptions

`kOTCfgManualConfig`	Use the IP address that is stored in the `fIPAddress` field.
`kOTCfgRARPConfig`	Obtain an address from a RARP server.

`kOTCfgBOOTPConfig`	Obtain an address from a BOOTP server.
`kOTCfgDHCPConfig`	Obtain an address from a DHCP server.
`kOTCfgMacIPConfig`	Configure TCP/IP to use MacIP.

Note
The TCP/IP control panel's "PPP Server" address acquisition method is actually implemented by setting `fConfigMethod` to `kOTCfgManualConfig` and setting `fIPAddress` to zero. ◆

Apple Remote Access Constants and Other Data Types

The following enumeration defines constants for the `version` field that appears in Apple Remote Access (ARA) structures:

```
enum {
    kOTCfgRemoteDefaultVersion  = 0x00020003,
    kOTCfgRemoteAcceptedVersion = 0x00010000
};
```

Constant descriptions

`kOTCfgRemoteDefaultVersion`
The version number with which new Remote Access preferences should be created.

`kOTCfgRemoteAcceptedVersion`
A version number that is acceptable for existing Remote Access preferences.

ARA Per-Connection Preferences Types

The following enumeration defines per-connection preference types for ARA:

```
enum {
    kOTCfgRemoteARAPPref            = 'arap',
    kOTCfgRemoteAddressPref         = 'cadr',
    kOTCfgRemoteChatPref            = 'ccha',
    kOTCfgRemoteDialingPref         = 'cdia',
    kOTCfgRemoteAlternateAddressPref= 'cead',
    kOTCfgRemoteClientLocksPref     = 'clks',
    kOTCfgRemoteClientMiscPref      = 'cmsc',
    kOTCfgRemoteConnectPref         = 'conn',
    kOTCfgRemoteUserPref            = 'cusr',
    kOTCfgRemoteDialAssistPref      = 'dass',
    kOTCfgRemoteIPCPPref            = 'ipcp',
    kOTCfgRemoteLCPPref             = 'lcp ',
    kOTCfgRemoteLogOptionsPref      = 'logo',
    kOTCfgRemotePasswordPref        = 'pass',
    kOTCfgRemoteTerminalPref        = 'term',
    kOTCfgRemoteUserModePref        = 'usmd',
    kOTCfgRemoteSecurityDataPref    = 'csec',
    kOTCfgRemoteX25Pref             = 'x25 ',
};
```

Constant descriptions

`kOTCfgRemoteARAPPref`
: The preference type for the `OTCfgRemoteARAP` (page 124) structure.

`kOTCfgRemoteAddressPref`
: The preference type for that contains the number to dial, in `'TEXT'` format, with a maximum of 255 characters. See also `OTCfgRemoteConnect` (page 127).

`kOTCfgRemoteChatPref`
: The preference type for that stores the log sin (chat) script, in `'TEXT'` format. See also `OTCfgRemoteConnect` (page 127).

`kOTCfgRemoteDialingPref`
: The preference type for the `OTCfgRemoteDialing` (page 131) structure.

`kOTCfgRemoteAlternateAddressPref`
: The preference type for the `OTCfgRemoteAlternateAddress` (page 123) structure.

`kOTCfgRemoteClientLocksPref`
: The preference type for the `OTCfgRemoteClientLocks` (page 125) structure.

`kOTCfgRemoteClientMiscPref`
: The preference type for the `OTCfgRemoteClientMisc` (page 127) structure.

`kOTCfgRemoteConnectPref`
: The preference type for the `OTCfgRemoteConnect` (page 127) structure.

`kOTCfgRemoteConnectPref`
: The preference type for the `OTCfgRemoteConnect` (page 127) structure.

`kOTCfgRemoteUserPref`
: The preference type that stores the user name as a Pascal string.

`kOTCfgRemoteDialAssistPref`
: The preference type for `OTCfgRemoteDialAssist` (page 130) structure.

`kOTCfgRemoteIPCPPref`
: The preference type for the `OTCfgRemoteIPCP` (page 132) structure.

`kOTCfgRemoteLCPPref`
: The preference type for the `OTCfgRemoteLCP` (page 133) structure.

`kOTCfgRemoteLogOptionsPref`
: The preference type for the `OTCfgRemoteLogOptions` (page 135) structure.

`kOTCfgRemotePasswordPref`
: The preference type for the `OTCfgRemotePassword` (page 135) structure.

`kOTCfgRemoteTerminalPref`
: The preference type for the `OTCfgRemoteTerminal` (page 138) structure.

`kOTCfgRemoteUserModePref`
: The preference type for the `OTCfgRemoteUserMode` (page 139) structure.

`kOTCfgRemoteSecurityDataPref`
: The preference type for a preference that stores data for a plug-in security module. The format of the data is determined by the security module. For external security modules the format of the data is untyped.

`kOTCfgRemoteX25Pref`
: The preference type for the `OTCfgRemoteX25` (page 140) structure.

ARA Global Preference Types

The following enumeration defines global preference types for Apple Remote Access (ARA)

```
enum {
    kOTCfgRemoteServerLocksPref     = 'slks',
    kOTCfgRemoteServerPortPref      = 'port',
    kOTCfgRemoteServerPref          = 'srvr',
    kOTCfgRemoteApplicationPref     = 'capt'
};
```

Constant descriptions

`kOTCfgRemoteServerLocksPref`
: Defined but not used by ARA.

`kOTCfgRemoteServerPortPref`
: The preference type for the `OTCfgRemoteServerPort` (page 137) structure.

`kOTCfgRemoteServerPref`
: The preference type for the `OTCfgRemoteServer` (page 136) structure.

`kOTCfgRemoteApplicationPref`
: The preference type for the `OTCfgRemoteApplication` (page 123) structure.

OTCfgRemotePPPConnectScript

The `OTCfgRemotePPPConnectScript` enumeration defines constants for use in the `chatMode` field of the `OTCfgRemoteConnect` (page 127) structure:

```
typedef UInt32 OTCfgRemotePPPConnectScript;
enum {
    OTCfgRemotePPPConnectScriptNone = 0,
    OTCfgRemotePPPConnectScriptTerminalWindow = 1,
    OTCfgRemotePPPConnectScriptScript = 2
};
```

Constant descriptions

`OTCfgRemotePPPConnectScriptNone`
: No connect script is configured.

`OTCfgRemotePPPConnectScriptTerminalWindow`
: A terminal window is used to make the connection.

`OTCfgRemotePPPConnectScriptScript`
: A chat script is used to make the connection.

OTCfgRemoteProtocol

The `OTCfgRemoteProtocol` enumeration defines constant s for use in the `serialProtocolMode` field of the `OTCfgRemoteConnect` (page 127) structure:

```
typedef UInt32 OTCfgRemoteProtocol;
enum {
    kRemoteProtocolPPP = 1,
    kRemoteProtocolARAP = 2,
    kRemoteProtocolAuto = 3
};
```

Constant descriptions

`kRemoteProtocolPPP` The protocol is PPP only.

`kRemoteProtocolARAP`
: The protocol is ARAP only.

`kRemoteProtocolAuto`
: Auto-detect PPP or ARAP (not supported in ARA 3.5 and later).

AppleTalk Remote Access Protocol (ARAP), an Apple Computer proprietary dialup AppleTalk protocol, was developed before the AppleTalk Control Protocol (ATCP, an implementation of AppleTalk over PPP) and is now deprecated.

OTCfgRemoteLogLevel

The `OTCfgRemoteLogLevel` structure defines values for use in the `logLevel` field of the `OTCfgRemoteLogOptions` (page 135) structure:

```
typedef UInt32 OTCfgRemoteLogLevel;
enum {
    kOTCfgRemoteLogLevelNormal  = 0,
    kOTCfgRemoteLogLevelVerbose = 1
};
```

Constant descriptions

`kOTCfgRemoteLogLevelNormal`	Normal ARA logging.
`kOTCfgRemoteLogLevelVerbose`	Verbose ARA logging.

OTCfgRemoteDialMode

The `OTCfgRemoteRedialMode` enumeration defines values for the `dialMode` field of the `OTCfgRemoteDialing` (page 131) structure:

```
typedef UInt32 OTCfgRemoteRedialMode;
enum {
    kOTCfgRemoteRedialNone,
    kOTCfgRemoteRedialMain,
    kOTCfgRemoteRedialMainAndAlternate
};
```

Constant descriptions

`kOTCfgRemoteRedialNone`	Do not redial if the an attempt to dial fails.
`kOTCfgRemoteRedialMain`	Redial the main number only if an attempt to dial fails.
`kOTCfgRemoteRedialMain`	Redial the main number and the alternate number if an attempt to dial fails.

OTCfgRemoteAnswerMode

The `OTCfgRemoteAnswerMode` enumeration defines constants for the `answerMode` field of the `OTCfgRemoteServerPort` (page 137) structure:

```
typedef UInt32 OTCfgRemoteAnswerMode;
enum {
    kAnswerModeOff = 0,
    kAnswerModeNormal = 1,
    kAnswerModeTransfer = 2,
    kAnswerModeCallback = 3
};
```

Constant descriptions

`kAnswerModeOff`	Answering is disabled.
`kAnswerModeNormal`	Answering is enabled.
`kAnswerModeTransfer`	Answering as a callback server. This value is not valid for the personal server.
`kAnswerModeCallback`	Answering enabled in callback mode.

OTCfgRemoteNetworkProtocol

The `OTCfgRemoteNetworkProtocol` enumeration defines constants for the `networkProtoFlags` field of the `OTCfgRemoteServerPort` (page 137) structure:

```
typedef UInt32 OTCfgRemoteNetworkProtocol;
enum {
    kOTCfgNetProtoNone = 0,
    kOTCfgNetProtoIP = 1,
    kOTCfgNetProtoAT = 2,
    kOTCfgNetProtoAny = (kOTCfgNetProtoIP | kOTCfgNetProtoAT)
};
```

Constant descriptions

`kOTCfgNetProtoNone`	Do not allow any connections.
`kOTCfgNetProtoIP`	Allow IPCP connections.

`kOTCfgNetProtoAT`	Allow AppleTalk connections (ATCP and ARAP).
`kOTCfgNetProtoAny`	Allow IPCP and AppleTalk connections.

OTCfgRemoteNetAccessMode

The `OTCfgRemoteNetAccessMode` enumeration defines constants for the `netAccessMode` field of the `OTCfgRemoteServerPort` (page 137) structure:

```
typedef UInt8 OTCfgRemoteNetAccessMode;
enum {
    kOTCfgNetAccessModeUnrestricted = 0,
    kOTCfgNetAccessModeThisMacOnly
};
```

Constant descriptions

`kOTCfgNetAccessModeUnrestricted`
The connected client can see other entities on the server's network.

`kOTCfgNetAccessModeThisMacOnly`
The connected client can only see entities on the server machine.

Modem Constants and Other Data Types

The following enumeration defines per-connection preference types for modem preferences:

```
enum {
    kOTCfgModemGeneralPrefs         = 'ccl ',
    kOTCfgModemLocksPref            = 'lkmd',
    kOTCfgModemAdminPasswordPref    = 'mdpw',
};
```

Constant descriptions

`kOTCfgModemGeneralPrefs`
: The preference type for the `OTCfgModemGeneral` (page 141) structure.

`kOTCfgModemLocksPref`
: The preference type for the `OTCfgModemLocks` (page 143) structure.

`kOTCfgModemAdminPasswordPref`
: Preference type for the preference that contains the administration password.

Modem Global Preference Types

The following enumeration defines the global preference type for modem preferences:

```
enum {
    kOTCfgModemApplicationPref      = 'mapt',
};
```

Constant descriptions

`kOTCfgModemApplicationPref`
: Preference type for the `OTCfgModemApplication` (page 142) structure.

OTCfgModemDialingToneMode

The `OTCfgModemDialogToneMode` enumeration defines constants for the `modemDialToneMode` field of the `OTCfgModemGeneral` (page 141) structure:

```
typedef UInt32 OTCfgModemDialogToneMode;
enum {
    kModemDialToneNormal,
    kModemDialToneIgnore,
    kModemDialToneManual
};
```

Constant descriptions

`kModemDialToneNormal`	Wait for dial tone.
`kModemDialToneIgnore`	Do not wait for dial tone.
`kModemDialToneManual`	Manual dialing.

AppleTalk Constants and Other Data Types

The following enumerations define masks for the `kOTCfgProtocolOptionsPref` preference when used in an AppleTalk entity:

```
enum {
    kOTCfgATalkNoBadRouterUpNotification        = 1 << 0,
    kOTCfgATalkNoAllNodesTakenNotification      = 1 << 1,
    kOTCfgATalkNoFixedNodeTakenNotification     = 1 << 2,
    kOTCfgATalkNoInternetAvailableNotification  = 1 << 3,
    kOTCfgATalkNoCableRangeChangeNotification   = 1 << 4,
    kOTCfgATalkNoRouterDownNotification         = 1 << 5,
    kOTCfgATalkNoRouterUpNotification           = 1 << 6,
    kOTCfgATalkNoFixedNodeBadNotification       = 1 << 7,
};
```

Each bit determines whether the AppleTalk protocol stack posts notifications for the corresponding network event.

Per-connection AppleTalk Preference Types

The following enumeration defines constants for per-connection AppleTalk preference types:

```
enum {
    kOTCfgATalkGeneralPref      = 'atpf',
    kOTCfgATalkLocksPref        = 'lcks',
    kOTCfgATalkPortDeviceTypePref= 'ptfm',
};
```

Global AppleTalk Preference Types

The following enumeration defines constants for global AppleTalk preference types:

```
enum {
    kOTCfgATalkNetworkArchitecturePref = 'neta'
};
```

Infrared Constants and Other Data Types

The following enumeration defines type codes for infrared preferences.

```
enum {
    kOTCfgTypeInfraredPrefs     = 'atpf',
    kOTCfgTypeInfraredGlobal    = 'irgo'
};
```

Constant descriptions

`kOTCfgTypeInfraredPrefs`
: Type code for a preference that contains per-connection infrared settings.

`kOTCfgTypeInfraredGlobal`
: Type code for a preference that contains global infrared settings.

OTCfgIRPortSetting

The `OTCfgIRPortSetting` enumeration defines constants for use in the `OTCfgIRPortSetting` field of the `OTCfgIRGeneral` (page 159) structure:

```
typedef UInt16 OTCfgIRPortSetting;
enum {
    kOTCfgIRIrDA = 0,
    kOTCfgIRIRTalk = 1
};
```

Constant descriptions

`kOTCfgIRIrDA`	Specifies the Infrared Data Association (IrDA) protocol.
`kOTCfgIRIRTalk`	Specifies the IRTalk protocol, Apple's proprietary Infrared protocol that was developed prior to the development of IrDA.

Glossary

AARP See AppleTalk Address Resolution Protocol.

Address Resolution Protocol (ARP) The Internet protocol that maps an IP address to a MAC address.

Apple Remote Access (ARA) The mechanism by which computers running Mac OS connect to remote sites.

AppleTalk Address Resolution Protocol (AARP) The protocol that reconciles addressing discrepancies in networks that support more than one set of protocols. For example, by resolving the differences between an Ethernet addressing scheme and the AppleTalk addressing scheme, AARP facilitates the transport of DDP packets over a high-speed EtherTalk connection.

AppleTalk Control Protocol (ATCP) The protocol that establishes and configures AppleTalk over PPP.

AppleTalk Data Stream Protocol (ADSP) A connection-oriented protocol that provides a reliable, full-duplex, byte stream service between any two sockets in an AppleTalk network. ADSP ensures in-sequence, duplicate-free delivery of data over its connections.

AppleTalk Session Protocol (ASP) A general-purpose protocol that uses ATP to provide session establishment, maintenance, and teardown, along with request sequencing.

AppleTalk Transaction Protocol (ATP) A transport protocol that provides loss-free transaction service between sockets. This service allows exchanges between two socket clients in which one client requests the other to perform a particular task and to report the results. ATP binds the request and response together to ensure the reliable exchange of request-response pairs.

ARA See Apple Remote Access.

area The highest level of organization in the Network Setup database. Areas contain entities. See also entity, named area, temporary area.

ASP See AppleTalk Session Protocol.

ATCP See AppleTalk Control Protocol.

ATP See AppleTalk Transaction Protocol.

Bootstrap Protocol The protocol used by a node to obtain the IP address of its Ethernet interfaces from another node on the network, thereby allowing the first node to boot without local storage media.

BOOTP See Bootstrap Protocol.

current area The area in which preferences are stored. Another name for the default area.

database reference A value that represents the open session with the Network Setup database.

Datagram Delivery Protocol (DDP) The network-layer protocol that is responsible for the socket-to-socket delivery of datagrams over an AppleTalk network.

DDP See Datagram Delivery Protocol.

default area The preferred name for the area in which preferences are stored. Another name for the current area.

DHCP See Dynamic Host Configuration Protocol.

DNS See Domain Name System.

Domain Name System (DNS) The system used on the Internet for translating the name of a network node to an IP address.

Dynamic Host Configuration Protocol A mechanism for assigning an IP address dynamically so that the address can be reassigned when the original assignee no longer needs it.

entity The unit of organization within an entity. See also global protocol entity, network connection entity, set entity.

global protocol entity An entity that contains information shared by all connections for a particular protocol.

ICMP See Internet Control Message Protocol.

International Telecommunication Union Telecommunication Standardization Sector An international body that develops worldwide standards for telecommunications technologies.

Internet Control Message Protocol (ICMP) A network-layer Internet protocol that reports errors and provides other information relevant to IP packet processing.

Internet Protocol (IP) 1) A set of protocols including TCP, UDP, and ICMP. IP provides features for addressing, type-of-service specification, fragmentation and reassembly, and security. 2) An IP network-layer protocol offering a connectionless internetwork service.

Internetwork Packet Exchange (IPX) A network-layer protocol used for transferring data between clients and servers.

IP See Internet Protocol.

IP Control Protocol (IPCP) The protocol that establishes and configures IP over PPP.

IPCP See IP Control Protocol.

IPX See Internetwork Packet Exchange.

ITU-T See International Telecommunication Union Telecommunication Standardization Sector.

LCP See Link Control Protocol.

Link Control Protocol (LCP) The protocol that establishes, configures, and tests data-link connections for use by PPP.

MAC address See media access control address.

MacIP A network-layer protocol that encapsulates IP packets in DDP packets for transmission over AppleTalk and that also provides proxy ARP services.

maximum transmission unit (MTU) The maximum number of bytes in a packet.

media access control address The six-byte data link layer address that is required for every device that connects to a network. Other devices in the network use MAC addresses to locate devices on the network and to create and update routing tables.

MTU See maximum transmission unit.

Name Binding Protocol (NBP) The AppleTalk transport-layer protocol that translates a character string name to the address of the corresponding socket client; NBP enables AppleTalk protocols to understand user-defined zones and device names by providing and maintaining translation tables that map names to corresponding socket addresses.

named area An area in which preferences are stored.

NBP See Name Binding Protocol.

network connection entity An entity that contains information for a single instance of a network protocol.

PAP See Printer Access Protocol.

PPP See Point-to-Point Protocol.

Point-to-Point Protocol (PPP) A protocol that provides host-to-network connections over synchronous and asynchronous circuits. PPP was designed to work with several network-layer protocols, such as IP, IPX, and ARA.

preference The unit of organization within an entity. Each preference corresponds to a structure containing the settings for a particular protocol.

preference type An `OSType` that identifies a particular preference.

Printer Access Protocol (PAP) The AppleTalk protocol that manages interaction between computers and print servers; PAP handles connection setup, maintenance, and termination, as well as data transfer.

proxy ARP A variation of the ARP protocol in which an intermediate device (such as a router) sends an ARP response to the requesting host on behalf of the node whose MAC address was requested.

RARP See Reverse Address Resolution Protocol.

Reverse Address Resolution Protocol (RARP) The Internet protocol that maps MAC addresses to IP addresses.

Routing Table Maintenance Protocol (RTMP) The AppleTalk protocol used to establish and maintain the routing information that is required by routers in order to route datagrams from any source socket to any destination socket on the network. Using RTMP, routers dynamically maintain routing tables to reflect changes in network topology.

RTMP See Routing Table Maintenance Protocol.

set entity An entity that is used to group global and network connection entities for a particular purpose. For example, a set entity can be used to group AppleTalk and TCP/IP configurations for a particular location, such as home or work.

TCP See Transmission Control Protocol/Internet Protocol.

TCP/IP See Transmission Control Protocol/Internet Protocol.

temporary area An area that is created when a named area is modified.

Transmission Control Protocol/Internet Protocol A connection-oriented transport-layer Internet protocol that provides reliable full-duplex data transmission.

User Datagram Protocol (UDP) A connectionless transport-layer Internet protocol that exchanges datagrams without acknowledgments or guaranteed delivery, requiring that error processing and retransmission be handled by other protocols.

UDP See User Datagram Protocol.

ZIP See Zone Information Protocol.

X.25 An ITU-T standard that defines how connections are maintained for remote terminal access and computer communications in public data networks.

Zone Information Protocol (ZIP) The AppleTalk session-layer protocol that maintains and discovers the network-wide mapping of network number ranges to zone names. NBP uses ZIP to determine which networks contain nodes that belong to a zone.

Index