# Open Transport Client Developer Note

PRELIMINARY
Revision 1.1b14
11/18/96

Abstract:

This document describes the application programming interfaces of the Open Transport endpoint and mapper libraries, which lets Macintosh developers write transport-independent network applications.





Apple Computer, Inc., thanks X/Open Company Limited for permission to include excerpts from its X/Open CAE Specification: X/Open Transport Interface (XTI) (Document Reference XO/CAE/92/600 or C192 ISBN 1-872630-29-4) throughout this manual.

# Contents

# About This Document

This document describes the application (client) programming interface of the Open Transport Library. Because the document is intended for all Open Transport client developers, it does not provide protocol-specific information.

## Related Documents

*Apple Shared Library Manager Developer's Guide*. This manual is part of the Apple Shared Library Manager distribution.

X/Open Developer's Specification (1990), Revised XTI (X/Open Transport Interface), ISBN# 1-872630-05-7.

Addendum to Revised XTI (X/Open Transport Interface), August 1991.

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

## Revision History

| | |
|---|---|
| 01/18/96 | Added a paragraph on kOTLookErr errors to the Rcv function |
| 08/11/95 | Updated to reflect revision 1.1b1 of OpenTransport |
| 01/14/95 | Updated to 1.0a3 |
| 10/10/94 | Revision 1.0d18/Updated document. |
| 09/16/94 | Revision 1.0d16(numbering change to match up with builds)<br>Updated document. |
| 07/18/94 | Revision 1.0d13(numbering change to match up with builds)<br>Revise entire document to reflect revised client interface. |
| 06/08/94 | Revision 1.0d12(numbering change to match up with builds)<br>Revise entire document to reflect revised client interface. |
| 02/14/94 | Revision 1.0d8 (numbering change to match up with builds)<br>Revise entire document to reflect revised client interface. |
| 11/15/93 | Revision 2.0<br>Revise entire document to reflect revised client interface.<br>Delete reference to non-client entities and concepts.<br>Delete references to obsolete client concepts, routines, and data structures.<br>Delete references to planned functionality.<br>Add new information, and supplement existing information.<br>Reorganize and reformat entire document.<br>Change name to "Open Transport".<br>.<br>.<br>. |
| 8/13/91 | Revision 1.0, second Draft<br>First distribution. |






## Format of Function Descriptions

Descriptions of Open Transport functions have the following format:

**FUNCTION** The name of the function, and a brief description.

**C INTERFACE** The C interface to the function.

**C++ INTERFACE** The C++ interface to the function.

**DESCRIPTION** A full description of what the function does. For functions having parameters, the description includes a table that describes which parameters are inputs and outputs to the function. Here is a key to the entries in the table:

| | |
|---|---|
| x | The parameter value is meaningful. |
| (x) | The content of the memory pointed to by the x pointer is meaningful. |
| ? | The parameter value is meaningful, but the parameter is optional. |
| (?) | The content of the memory pointed to by the ? pointer is optional. |
| / | The parameter is meaningless. |
| = | The parameter after the call keeps the same value as before the call. |

**VALID STATES** A list of the states that an endpoint is allowed to be in when this function is called. Refer to Appendix B for a list and descriptions of valid endpoint states.

**RESULT CODES** A list of result codes that the function can return. Refer to Appendix D for a list and descriptions of Open Transport result codes.

**SEE ALSO** A list of related functions, if any.

# Introduction to Open Transport

Open Transport is a set of low-overhead interfaces that will become the standard interfaces for Macintosh networking and communications. By using the Open Transport interfaces, applications (called *clients*) can operate in a transport-independent manner. (In Open Transport, the term *transport* refers to any protocol that moves, or tranports, data.) The Open Transport client interfaces are a superset of the XTI interface from X/Open , a consortium of UNIX vendors. XTI is a superset of TLI, a UNIX standard interface.

Using the Open Transport client interface does not, by itself, ensure transport independence. Some applications are tightly coupled with a particular protocol. (For example, the current implementation of AppleShare on Macintosh computers is tightly coupled to the AppleTalk ASP protocol.) These applications often make use of particular protocol features not available in other protocols. The Open Transport Library lets such applications use all the features provided by a particular protocol through options or private interfaces. The penalty for using such protocol-specific features is, of course, that such applications are not transport independent.

The Open Transport Library uses the same option management scheme as the XTI interface, for which X/Open has already defined the options of TCP/IP, OSI, NetBIOS, and other network systems. This ensures that, even if an application is tied to a specific protocol, the application will be transportable across other implementations of the protocol, because the options have been standardized.

The rest of this introduction describes the main features of Open Transport.

## Transport Independence

One of the goals of Open Transport is to separate the evolution of AppleTalk services from the evolution of AppleTalk transport protocols. This is the keystone of Apple's approach toward "open systems." Clients can use transport protocol families other than AppleTalk yet still have access to AppleTalk's superior desktop services, such as file sharing and printing. Within this framework, the main goal of Open Transport is to enable the creation of transport-independent client/server and peer-to-peer applications. For example, it should be possible to add a new networking transport protocol, such as OSI transport, and have file sharing operate over it without modification.

## Transport Transparency

Transport transparency is similar to transport independence but goes further toward separating an application from the transport below it. Transport-independent interfaces allow applications to use similar transports interchangeably, but the application is still responsible for telling the system which protocol stack is going to be used. Transport transparency implies that the application does not choose the protocol to be used.

The X/Open XTI interface uses a text string to name the desired protocol stack. An application using the XTI interface is responsible for providing the text string. Open Transport provides the same mechanism. Thus, the Open Transport interface cannot, alone, provide transport transparency, because an application must supply this information. However, when combined with a Chooser/Browser and/or address book, an application can become completely transport-transparent. When an entity is selected using a Chooser/Browser or address book, information is returned that an application just turns around and gives to the Open Transport interface. This information allows the Open Transport Library to choose or build the correct protocol stack, and the information may contain configuration settings for communicating with the selected entity.

In summary, the Open Transport Library provides transport independence. When combined with a Chooser/Browser and/or address book, it will provide transport transparency.






## Transaction Protocols

A transaction protocol is one in which a request is issued, and a response to that request is received. Most AppleTalk services (such as AppleShare, printing, and many third-party products) are based one or both of Apple's transaction protocols (ATP and ASP). X/Open's XTI interface, however, makes no provision for transaction protocols. For this reason, Open Transport extends XTI by adding transaction protocols. The additional functions and data structures are implemented in the same style as all of the other functions, so that the transaction extensions do not appear to be "tacked on."

## Operating Environment

The Open Transport Library is designed to be an appropriate interface for any underlying operating system. It does not depend on multitasking for its operation or interface, but it can take advantage of multitasking in an environment that provides it. The framework provided by the Open Transport Library for use by protocol implementations is designed to be independent of the underlying operating system. The first release of the Open Transport Library works in both 68000-family (68030 and 68040 CPU) Macintosh and PowerPC Macintosh environments.

## Client Programming Interfaces

The Open Transport Library has three related client programming interfaces: XTI-style, preferred C, and preferred C++.

- **XTI-style interface.** The C-language XTI calls, plus some Open Transport extensions, form a group of functions referred to in this document as the "XTI-style" interfaces. These are not the preferred interfaces on the Macintosh because of the way they handle errors—through the use of a global variable. The names of these functions start with "`t_`": for example, `t_bind` and `t_accept`. These functions are provided solely to ease porting of existing XTI client code.

- **C functions with Macintosh conventions for names and error handling.** For each XTI-style function, there is a corresponding preferred-C function These functions are referred to in this document as the "Preferred C" interfaces. These functions have names like `OTBind` and `OTAccept`.

- **A C++ class (`TEndpoint`) and its member functions.** These functions are referred to in this document as the "Preferred C++" interface. For each preferred-C routine, there is a corresponding member function of the `TEndpoint` class with the same name.

The preferred-C and preferred-C++ interfaces have different error codes than XTI does. XTI error codes are small positive integers, but such values would be inconsistent with the rest of the Macintosh Toolbox. For the Open Transport preferred interfaces, there are a set of constants defined that map one-to-one with the XTI error constants.

> **IMPORTANT:** This document refers to Open Transport functions by their preferred-C++ names, except where differentiating between C and C++ or between XTI and Open Transport.

Finally, XTI defines several structures, pointers to which are used as parameters to the XTI functions. Examples of these structures are `t_netbuf`, `t_info`, `t_bind`, and `t_call`. The Open Transport preferred interfaces use structures with more Macintosh-like names such as `TNetbuf`, `TEndpointInfo`, `TBind`, and `TCall`. In general, these structures are identical to their XTI counterparts.

## Open Transport and Interrupt Routines

Open Transport also defines many support routines that clients can use to deal with the communications environments. These are described in the Utility routine section toward the end of this document.

The Open Transport API is intended to provide high-performance communications services to client applications. In keeping with this goal, **Open Transport functions may never be called at interrupt time**. This includes any interrupt routine from an external device, VBL tasks, or Time Manager. Open Transport functions may only be called at primary task time (also called "System Task" time, or at Deferred Task time (also called Secondary Interrupt level) scheduled by using either the Open Transport function OTScheduleDeferredTask, or by using the system _DTInstall trap.

In order to support calling at primary interrupt time, Open Transport would have to be able to turn interrupts on and off to protect critical resources. On PowerPC machines, this requires a costly mixed-mode switch. Open Transport provides the functions OTCreateDeferredTask, OTScheduleDeferredTask, and OTDestroyDeferredTask to make it very easy for clients to defer their operations to deferred task time without using confusing paramater blocks. Please use them.






# Getting Started

This section describes the main steps you must follow to create an Open Transport client in C or C++.

**IMPORTANT:** On Macintosh computers, the Open Transport libraries are implemented using the Apple Shared Library Manager. Thus, Open Transport clients are also clients of the Shared Library Manager, and must obey the rules of its environment. (Refer to the *Apple Shared Library Manager Developer's Guide* for more information.)

## Determining if Open Transport is installed

There are two ways to determine if Open Transport is present. The first is to call `InitOpenTransport`. If it returns an error, Open Transport is not present or cannot be loaded. The second is to use the `Gestalt` function. The selector for Open Transport is 'otan'. If `Gestalt` returns no error and the `response` parameter returned is non-zero, Open Transport is installed.

## Using Open Transport From Stand-Alone Code

If your client is a stand-alone code segment or code fragments, it must perform the following steps before calling any Open Transport functions for the first time:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. Establish an A5 world. One possible way of doing this is described in the *Apple Shared Library Manager Developer's Guide.* For code fragments, the global world is already established for you, and this step is not necessary.
3. Call the routine `InitOpenTransport` Be sure to have linked with OpenTransportExtn.o (for 68K) or OpenTransportExtnPPC.o (for PowerPC). If it returns an error, Open Transport is not available. If you are also using the Apple Shared Library Manager, then you should make the `InitLibraryManager` call prior to calling `InitOpenTransport`.

```
pascal OSStatus InitOpenTransport()
```

After performing these steps, a standalone-code client must check that it is in its own A5 world **each** time it calls an Open Transport function. Finally, before exiting, a stand-alone client must call the routine `CloseOpenTransport`. If you are also using the Apple Shared Library Manager, you should call `CleanupLibraryManager` after calling `CloseOpenTransport`.

## Using Open Transport From Applications

If your client is an application, not a stand-alone code segment or code fragment library, it must perform these steps to use the Open Transport Library:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. Call the routine `InitOpenTransport` before calling the Open Transport Library for the first time. Be sure to hav elinked with OpenTransportApp.o (for 68K) or OpenTransportAppPPC.o (for PowerPC). If you are also using the Apple Shared Library Manager, then you should make the `InitLibraryManager` call prior to calling `InitOpenTransport`.
3. (optional) Call the routine `CloseOpenTransport` before exiting.

Unlike stand-alone code segments, applications do not need to explicitly establish an A5 world before calling the Open Transport Library nor reset their A5 world before each call to an Open Transport function - the Macintosh runtime takes care of that.

### Using Open Transport From C Clients

Although most Open Transport functions operate the same way, whether called from C or C++, the names of equivalent Open Transport calls differ slightly between the two languages. The difference is that, in C, Open Transport function names have the prefix "`OT`". For example, calling `OTBind` from C is the equivalent of calling `Bind` from C++. Except where differentiating between the two languages, this document omits the "`OT`" prefix when referring to Open Transport functions.

### Using Open Transport From C++ Clients

An unfortunate fact of life is that different compiler vendors handle C++ dispatching differently. The C++ objects for Open Transport (and the Apple Shared Library Manager) conform to the MPW CFront "SingleObject" dispatching convention for 68K Macintosh, and to the MPW PPCC dispatching convention for PowerPC. Every attempt will be made to make the header files and C++ interfaces usable by other compilers (for instance, we force pascal or _cdecl calling conventions on all of the methods of the classes to allow Symantec C++ to use our classes. The intent of the Open Transport team is to support SOM (IBM's System Object Model) when it becomes available.

# About Providers

To understand how Open Transport clients work, you must first understand providers. This section describes what providers are.

A provider is a software entity that provides some kind of data-oriented service. That service might be implementing a networking protocol, encrypting data, filtering data, or some other data-oriented services. Providers are implemented by modules which can be layered to provide an arbitrarily complex service for clients. For example, an encryption module can be placed above the AppleTalk Data Stream Protocol (ADSP), which is placed above an Ethernet module. This combination would provide a networking stream of data that was secure from "snooping" on the network.

A client interfaces with a provider by means of a handle called an `ProviderRef` (for C++ users, this is the class `TProvider`). Conceptually, a `ProviderRef` is similar to a file handle or a driver reference number. It provides the association between the function called by the client and the specific provider which is to act on that function.

All providers support a basic amount of functionality. This includes functions to make the provider synchronous or asynchronous, setting the blocking behavior, install and remove notification routines, send IOCTL commands, cancel outstanding synchronous calls, and close.

Open Transport supplies several different types of providers. A provider called a mapper (referenced by a `MapperRef`) provides name-to-address translation services. A provider called an endpoint (referenced by an `EndpointRef`) provides a service to create connections and move data from one machine to another. Each of these additional providers support the basic functionality, and in addition, support an additional set of functions that are unique to the type of provider.






It is important to realize that references to other types of providers (e.g. `EndpointRef`) can be used in any function that requires a `ProviderRef` as a parameter. For instance, the function `OTIoctl` (which allows protocol-specific commands to be sent to the communications provider) requires a `ProviderRef` as it's first parameter. It is perfectly valid to pass an `EndpointRef` to this function (The converse is not true, however. If a function requires an `EndpointRef`, never pass it a `ProviderRef`. This may cause a crash!)

# Specifying provider services

Clients invoke provider services by using an open routine to first obtain a reference to the desired provider. This document describes two of these: `OTOpenEndpoint` and `OTOpenMapper`. Protocol families may provide other services, with each of these other services having their own unique open routines.

All open routines take a pointer to an `OTConfiguration` structure as a parameter. This structure is not defined by the Open Transport header files. The only way to create one is to call `OTCreateConfiguration` (see the Utility functions section), passing it a string describing the provider service desired.

In its simplest form, this string is just the name of a protocol (e.g. "`ddp`", "`tcp`", or "`dnr`"). Open Transport will use default rules to supply the rest of the information (which data link to use, for instance). In its full form, this string can be a comma-separated list of provider names, with option values specified by enclosing them in parenthesis after the name of the module to which they apply. For instance,

```
"nbp,ddp,ltlkB"
```

describes an NBP provider layered above a DDP provider layered on Printer port LocalTalk, and

```
"adsp,ddp(Checksum=1),enet"
```

describes an ADSP provider layered above a DDP provider (with checksumming turned on) layered above the default ethernet card.

Open Transport will attempt to use default rules to complete specifications that are incomplete. For instance, the specification "adsp,enet" is incomplete, since ADSP cannot run layered directly on top of ethernet. Open Transport understands this, and automatically puts the DDP layer between ADSP and ethernet.

The various names of providers are provided as constants in the appropriate header files for the protocol families.

### Specifying ports

Open Transport provides a standard naming scheme for describing various hardware ports on a system. Every port on the machine is described by the information in an OTPortRecord structure:

```
// values for the fInfoFlags field of OTPortRecord
enum
{
        kOTPortIsDLPI               = 0x00000001,
        kOTPortIsTPI                = 0x00000002,
        kOTPortIsNotShareable       = 0x00002000,
        kOTPortIsSystemRegistered   = 0x00004000,
        kOTPortIsPrivate            = 0x00008000,
        kOTPortIsAlias              = 0x80000000
```

```
};

// values for the fPortFlags field of OTPortRecord
enum
{
        kOTPortIsActive             = 0x00000001,
        kOTPortIsDisabled           = 0x00000002,
        kOTPortIsClosed             = 0x00000004
};

struct OTPortRecord
{
        OTPortRef           fRef;
        UInt32              fPortFlags;
        UInt32              fInfoFlags;
        UInt32              fCapabilities;
        size_t              fNumChildPorts;
        OTPortRef*          fChildPorts;
        char                fPortName[kMaxProviderNameSize];
        char                fModuleName[kMaxModuleNameSize];
        char                fSlotID[kMaxSlotIDSize];
        char                fResourceInfo[kMaxResourceInfoSize];
        char                fReserved[164];
};
```

fRef is a 32-bit value that is unique to this port.

fPortFlags are operational flags associated with the port. They describe the current status of the port.

fInfoFlags are configuration flags associated with the port.

fCapabilities are the capability flags for the device. These indicate what kind of framing the port can handle, or enumerate other types of capabilities by using bits in the flag. The meaning of the bits is specific to the device type of the port.

fNumChildPorts This is the number of devices that are configured directly below this device. Typically, this number is 0 or 1. For instance, a "modem" device normally has a serial port device directly below it.

fChildPorts This is an array of ports containing the number of entries specific by the *fNumChildPorts* field.

fPortName is a zero-terminated string that is unique to this port. This name may be used by OTCreateConfiguration to specify this port uniquely.

fModuleName is the name of the provider module that Open Transport loads. This is not normally relevant to Open Transport clients.

fSlotID is a 0-terminated string. It contains a string describing the slot that the port resides in. For many devices, this string is empty, indicating that it is up to the application to determine how to identify the slot. In the current implementation of OpenTransport, only PCI devices have a non-empty string in this field.

fResourceInfo is a 0-terminated string that describes a shared library that can handle configuration things for the device. The full description of this field is being worked out, and will be described in a separate document.






The header file OpenTransport.h defines a number of utility functions for creating and extracting information from OTPortRef values (See the Utility Functions section later in this document for a description of these functions).

The header file OpenTptLinks.h contains a list of the current valid device types:

```
enum
{
    kOTADEVDevice               = 1,        /* An Atalk ADEV            */
    kOTMDEVDevice               = 2,        /* A TCP/IP MDEV            */
    kOTLocalTalkDevice          = 3,        /* LocalTalk                */
    kOTIRTalkDevice             = 4,        /* IRTalk                   */
    kOTTokenRingDevice          = 5,        /* Token Ring               */
    kOTISDNDevice               = 6,        /* ISDN                     */
    kOTATMDevice                = 7,        /* ATM                      */
    kOTSMDSDevice               = 8,        /* SMDS                     */
    kOTSerialDevice             = 9,        /* Serial                   */
    kOTEthernetDevice           = 10,       /* Ethernet                 */
    kOTSLIPDevice               = 11,       /* SLIP Pseudo-device       */
    kOTPPPDevice                = 12,       /* PPP Pseudo-device        */
    kOTModemDevice              = 13,       /* Modem Pseudo-Device      */
    kOTFastEthernetDevice       = 14,       /* 100 MB Ethernet          */
    kOTFDDIDevice               = 15,       /* FDDI                     */
    kOTATMLANEDevice            = 16,       /* ATM LAN emulation        */
    kOTATMSNAPDevice            = 17        /* ATM SNAP emulation       */
};
```

New device types should not be arbitrarily added. Please contact Apple Computer to obtain a new, unique device type (Use the OPENTPT e-mail address mentioned on the first page of this document).

With this information, a client can create OTPortRefs for many of the common hardware devices. For instance

```
OTPortRef ref = OTCreatePortRef(kOTNuBus, kOTEthernetDevice, 10, 0);
```

defines the OTPortRef for an Ethernet card in slot 10 of the NuBus. Note that slot numbers are physical; that is, they are the slot number returned by the SlotManager and NOT the slot number that are seen in various network configuration applications. For cards in a PCI bus, it is not possible, a priori, to create a port ref that corresponds to a know card, so applications will need to iterate through the port registry to find appropriate PCI ports.

Each port on a machine has three different names associated with it: the device name, the port name, and the module name.

The device name for a port is associated with the device type of the port . All ports on a machine that are the same device type have the same device name.. These names are defined in the header files OpenTptAppleTalk.h (for LocalTalk), OpenTptSerial.h (for serial), and OpenTptLinks.h (for all the rest).

All ports on the machine also have a unique port name. This name uniquely identifies the port without using any location information. For instance, "ltlkA" is the unique identifier given to LocalTalk on the serial port, and "ltlkB" is the unique identifier given to LocalTalk on the modem port. This name must always be used in the path string for `OTCreateConfiguration` to uniquely identify a port.

The last name associated with a port is the module name. For Open Transport clients, this name is not necessary to know. It is needed by Open Transport to know which provider module to load when the port is specified, and is supplied to clients for informational purposes.

Open Transport allows clients to use just a device name to specify a port. In this case, Open Transport will use the FIRST device of that type that is registered and available. For most devices, this means the motherboard device if it exists. Otherwise, it is the first slotted device that was registered.

Open Transport provides functions to iterate through all of the ports on the machine, and determine what type of port they are (ethernet, serial, etc.), and what their names are by returning the OTPortRecord information. See the Utility function section near the end of this document for functions which can be used to find out information about ports.

## Modes of Operation

Open Transport providers can operate in either of two modes: synchronous or asynchronous. The mode of a provider affects when calls to certain provider functions return to the calling client.

When a client calls a provider function in a synchronous mode, control does not return to the client until the function has been completed. In contrast, when a client calls a provider function in asynchronous mode , control returns to the client immediately. Later, when the function is complete, an event is sent to notify the client (NOTE: Be aware that it is possible for the notification event to be received by the client BEFORE the asynchronous function call returns to the caller).

By default, Open Transport providers operate in whatever mode they were opened in (i.e. if the provider was opened synchronously, the provider defaults to synchronous mode, otherwise it defaults to asynchronous mode). A client can, however, change the provider's mode of operation. All subsequent provider calls operate in the new mode.

## Blocking

Blocking is he second attribute that a client can apply to a provider to govern how the client interacts with the provider. In order for a provider to service a client request, the provider may be able to deal with the request immediately, or it may have to queue the request up and deal with it later. When the provider is in non-blocking mode, if the request needs to be queued to be dealt with later, a kEAGAINErr is returned to the caller. This allows the client the flexibility to reissue the command again later. The non-blocking mode is more important to synchronous-mode clients, but it also can affect asynchronous-mode clients.

Blocking also affects how flow control is handled for endpoints on synchronous calls. In blocking mode, a send request will wait for flow control to lift, then complete the send. In non-blocking mode, if flow control is on, a kOTFlowErr or a number indicating only a partial send will be returned to the client (a T_GODATA or T_GOEXDATA event will be sent to the client when flow control lifts).

When a provider is created, Open Transport defaults to non-blocking mode for all endpoints. After that, the client has complete control over both the blocking/non-blocking and synchronous/asynchronous behavior of the provider.

## AckSends

Open Transport provides a mode for sending data called "AckSends". In this mode, Open Transport does not necessarily copy the data to be transmitted. Instead, the client is notified when the memory is no longer being used by the provider via a T_MEMORYRELEASED event.






## Provider Events

In Open Transport, as in XTI, providers communicate with clients by issuing events. Open Transport includes a set of XTI-defined **asynchronous events** that signal occurrences such as the arrival of data. But XTI does not define how clients handle these events; rather, event handling is operating-system-dependent. For this reason, Open Transport provides its own mechanism for handling events. Also, because Open Transport is a superset of XTI, it defines additional asynchronous events. Open Transport further extends XTI by defining **completion events**, which inform a client when an asynchronous operation is finished. The names of completion events end in “COMPLETE”—for example, T_BINDCOMPLETE.

To handle asynchronous events and completion events, an Open Transport client must provide a single event-handling routine that the provider can call when events occur. Optionally, the client can also continuously poll the provider for asynchronous events. (Clients cannot poll providers for completion events.) In general, the preferred method for handling all Open Transport events is to provide an event-handling routine. For more information about handling events, refer to the section “Handling Events.”

# Provider Functions

This section describes the provider functions that clients can call. The descriptions are grouped by task, and each group is preceded by a brief introduction. The functions are described using their Open Transport preferred-C++ names. Appendix E lists the corresponding XTI-style names.

## Result codes

Most Open Transport functions return an OSStatus value that indicates whether the operation succeeded or failed.

For synchronous function calls, the result code indicates whether the operation succeeded or failed. If `kOTNoError` is returned, the operation succeeded. If some other value is returned, the operation failed. The value will indicate the cause of failure.

For asynchronous function calls, if the result code is `kOTNoError`, then the operation was successfully started. When it completes, your notification function will be called with an event code to indicate which operation completed and a result code indicating whether the operation was successful or failed. If an asynchronous function call returns a different value, then the operation has failed before it was started, and your notification function will not be called.

In the function descriptions which follow, an attempt is made to enumerate the result codes returned by each function. However, the list may not be complete, and a client should never assume that other result codes will not be returned. In general, if you receive a result code you do not understand, your code should assume the worst.

Also, the error code `kEBADFErr` is returned if the `ProviderRef` supplied to the function call is incorrect. This fact is ignored in the descriptions (i.e. some function calls indicate that they return no error. This is true if the supplied `ProviderRef` is valid).

Every function in Open Transport can return the following result codes that are not enumerated in the function lists:

| | |
|---|---|
| `kOTBadSyncErr` | A synchronous call was attempted at interrupt or deferred task level, or an Open Transport call was made at primary interrupt time. NOTE: Open Transport cannot always detect these conditions, so your code should not rely on getting this error if it calls Open Transport at the wrong time. If this situation is undetected by Open Transport, the machine could crash. |
| `kENOMEMErr` | There is not enough available memory to complete the request. |
| `kENOSRErr` | There are not enough system resources to complete the request. Typically, this means that there are no more STREAMS messages available. |
| `kEAGAINErr` | An endpoint or mapper is in non-blocking mode, and Open Transport would have to block to complete the request. This is also referred to as the `kEWOULDBLOCKErr` in the documentation (they are the same error code). |
| `kOTProtocolErr` | An unspecified protocol error occurred. This is usually fatal. Normal recovery is to close the provider. |






| | |
|---|---|
| `kEBADFErr` | This error can only be returned by the C interfaces. It indicates that the `ProviderRef` supplied to the function was not a valid reference. |
| kOTClientNotInittedErr | The client has not called InitOpenTransport. |
| `kOTStateChangeErr` | This error normally only occurs when an asynchronous function is executed. It is very similar to a `kOTOutStateErr`, but instead of indicating that the endpoint is in the wrong state, it indicates that either the state is in the process of changing, or that some function on the endpoint is currently being executed that is incompatible with the requested function. For example, this could occur if a `SndUData` function is called, and before it is completed, an Unbind is issued from a deferred task routine. It is not correct to return a `kOTOutStateErr`, since the endpoint is in the correct state to do an `Unbind`. However, until the `SndUData` is complete, Open Transport cannot issue the `Unbind` call. |

## Handling Events

This section describes how an Open Transport client can handle provider events. For a list and description of the events that Open Transport defines, refer to Appendix C.

**Note:** In addition to the general events defined by Open Transport, some providers issue additional protocol-dependent events. These events are described in the documentation for each provider. Transport independent clients, however, should ignore such events. Likewise, all clients should ignore any event that they do not recognize.

## Using Notifier Routines

When using the Open Transport Library, the preferred way to handle events is to install a notifier routine. A notifier routine (a “notifier,” for short) is a callback routine. The Open Transport Library uses it to call back into a client, notifying the client that an event has occurred. Notifier routines are written by a client and installed on a provider after it is opened. They are also used in asynchronous Open routines.

A notifier function is a ‘C’ style function with the following prototype:

```
pascal void NotifierRoutine(void* contextPtr, OTEventCode code,
                            OTResult result, void* cookie);
```

To install a notifier routine on an endpoint, a client calls the function `OTInstallNotifier`; to remove the notifier, the client calls `OTRemoveNotifier`. Both of these functions are described in the sections that follow.

A notifier function is of type `void` and has the following parameters:

| | |
|---|---|
| `contextPtr` | The value of this `void*` pointer is specified by the client when the notifier is installed. Typically, the client will use this parameter to recover some kind of context information. It could be a `ProviderRef`, or a pointer to a data structure that the client set up (typically, this data structure has the `ProviderRef` in it somewhere). |

`code` This is the event code; it identifies what kind of event occurred. The event codes are defined in the header file `OpenTransport.h`.

`result` This parameter has no meaning for most asynchronous events. For the completion events, it contains the result code of the completed function.

`cookie` This parameter has different meanings depending upon the event code. For all of the standard XTI events and some of the Open Transport event extensions, this pointer has no meaning. Appendix C shows all of the Open Transport events and the meaning of the cookie parameter for each event.

There is a "C" typedef for a pointer to a notification routine, `OTNotifyProcPtr`, that looks like:

```
typedef pascal void (*OTNotifyProcPtr)(void* contextPtr, OTEventCode code,
                        OTResult, void* cookie);
```

On 68000-family Macintosh computers, restoring the A5 world in a notification routine is not necessary; it is done automatically. The value of A5 is saved when the client calls the function `InstallNotifier` (described later in this section) and is restored every time the notification routine is called. If your development environment uses some other register for context, then your client must save and restore this context.

**Note:** A notification routine may be called at deferred-task time, and it may be called reentrantly. On Macintosh computers running System 7, a notification routine has all the same restrictions as any other interrupt-level callbacks, such as VBLs, Time Manager tasks, and Device Manager completion routines. An attempt is made in Open Transport to queue calls to a client's notification routine to prevent reentrancy and keep the processor stack from growing, but this behavior is not guaranteed. Clients should be prepared and write their notification routine defensively.






# InstallNotifier

**FUNCTION**

`InstallNotifier` Give a provider a notification routine to use for notifying a client of events.

**C INTERFACE**

```
OSStatus    OTInstallNotifier(ProviderRef ref, OTNotifyProcPtr* proc,
            void* contextPtr);
```

**C++ INTERFACE**

```
OSStatus    TProvider::InstallNotifier(OTNotifyProcPtr* proc, void*
            contextPtr);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `proc` | x | / |
| `contextPtr` | x | / |
| `notifier` (C++ only) | x | / |

`InstallNotifier` gives a provider a notification routine for passing events from the provider back to the client.

In C, the `ref` parameter must refer to a valid provider. In C++, `InstallNotifier` is a member function of the `TProvider` class, so no explicit provider reference is required.

The `proc` parameter is a pointer to a notification routine. This routine must be a "C" routine. If you are programming in C++, do not take the address of a member function of a class and pass that as the notification routine, unless that routine is declared static.

The typedef for the `proc` function pointer is:

```
typedef void (*OTNotifyProcPtr)(void* ref, OTEventCode event, OTResult err, void*
cookie);
```

The `contextPtr` parameter is not interpreted by the Open Transport Library; rather, it is saved internally and passed as the first parameter to the notification routine.

When an event occurs, the client's notification routine is called, and the same pointer passed to `InstallNotifier` is passed to the notification routine as the first parameter. The client typically will use this to hold some kind of context. The provider will not reference the data at `ref` nor modify it.

The `cookie` parameter that is passed to the notification routine has a meaning that depends upon the event code. For many event codes, `cookie` has no meaning. For completion events such as T_REPLYCOMPLETE, where a client can have multiple outstanding occurrences of the same type of call, `cookie` is used to help the client determine which particular call completed. The meaning of `cookie` in each context is described in each affected call's description of asynchronous operation.

If you try to install a notifier on a provider that has one already, `InstallNotifier` returns the error code `kOTAccessErr`. To remove a notification routine from a provider, call the `RemoveNotifier` function.

**RESULT CODES**

`kOTAccessErr`

**SEE ALSO**

`GetNotifier`, `RemoveNotifier`






# RemoveNotifier

**FUNCTION**

`RemoveNotifier` Remove the notifier from a provider.

**C INTERFACE**

```
void        OTRemoveNotifier(ProviderRef ref);
```

**C++ INTERFACE**

```
void        TProvider::RemoveNotifier();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref (C only)` | x | / |

`RemoveNotifier` removes the notification routine currently installed on the specified endpoint. If no notifier is installed, this function does nothing.

The parameter `ref` specifies the endpoint whose notification routine is to be removed.

**RESULT CODES**

none

**SEE ALSO**

`GetNotifier, InstallNotifier`

## Setting the Mode of Operation

At creation, providers operate in the mode they were created in (i.e. If the provider was opened synchronously, it will be in synchronous mode. If it was opened asynchronously, it will be in asynchronous mode). Clients can find out the current mode of a provider and set it by using the routines described in this section.

# SetSynchronous

**FUNCTION**

`SetSynchronous` Place a provider into synchronous mode.

**C INTERFACE**

```
OSStatus    OTSetSynchronous(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::SetSynchronous();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`SetSynchronous` sets the operational mode of the provider to synchronous. To find out whether an endpoint is synchronous, use the function `IsSynchronous`.

This call can be made at any time and will always succeed.

**RESULT CODES**

`None`

**SEE ALSO**

`IsNonBlocking, SetBlocking, SetNonBlocking, SetAsynchronous, IsSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper`






# SetAsynchronous

**FUNCTION**

`SetAsynchronous` Place a provider into asynchronous mode.

**C INTERFACE**

```
OSStatus    OTSetAsynchronous(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::SetAsynchronous();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`SetAsynchronous` sets the operational mode of the endpoint to asynchronous. To find out whether an endpoint is asynchronous, use the function `IsSynchronous`.

This call can be made at any time and will always succeed.

**RESULT CODES**

`None`

**SEE ALSO**

`IsNonBlocking, SetBlocking, SetNonBlocking, SetSynchronous, IsSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper`

# IsSynchronous

**FUNCTION**

`IsSynchronous` Determine if a provider is in the synchronous mode of operation.

**C INTERFACE**

```
Boolean     OTIsSynchronous(ProviderRef ref);
```

**C++ INTERFACE**

```
Boolean     TProvider::IsSynchronous();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`IsSynchronous` returns true if the endpoint is in synchronous mode, or returns false if it is in asynchronous mode.

**RESULT CODES**

`None`

**SEE ALSO**

`IsNonBlocking, SetBlocking, SetNonBlocking, SetAsynchronous, SetSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper`






## Controlling Operations on a Provider

This section describes provider functions which can be used by clients to control operations on providers.

# OTCloseProvider

**FUNCTION**

`CloseProvider` Close and delete a provider when it is no longer needed.

**C INTERFACE**

```
OSStatus    OTCloseProvider(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::Close();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`CloseProvider` closes a provider when it is no longer needed. The `CloseProvider` call is always synchronous to the caller, but asynchronous in operation. When the call returns, if no error occurred the ProviderRef is no longer valid and should not be used.

`CloseProvider` may be called at any time (except at primary interrupt level, of course). However, if there are outstanding asynchronous calls, they will never complete, so be prepared for this. Any outstanding synchronous call will be completed with a result code of kOTCanceledErr.

The blocking/non-blocking status of the provider also governs what happens when the provider is closed. In non-blocking mode, closing the provider will flush all outgoing commands in the stream, and immediately close the provider. In blocking-mode, the stream will be given up to 15 seconds per module to allow outgoing commands and data to be processed, then the stream will be closed. However, as far as the caller is concerned, the provider is closed immediately.

**RESULT CODES**

`None`

**SEE ALSO**

`OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper, AckSends, SetBlocking, SetNonBlocking`

# SetBlocking

**FUNCTION**

`SetBlocking` Block (wait) for completion of outstanding I/O requests on a provider.

**C INTERFACE**

```
OSStatus    OTSetBlocking(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::SetBlocking();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`SetBlocking` governs how I/O requests proceed over the specified provider.

After a client calls `SetBlocking`, input requests on the specified provider do not complete unless they are satisfied. For example, if the client issues a `Connect` request in blocking mode, the Connect will complete even if there is contention for resources. However, in non-blocking mode, if there is contention for a STREAMS resource, the command will complete with a kEAGAINErr, and the command needs to be reissued again later when there is no contention. For example, if the client issues an asynchronous IOCTL command that is not yet complete, and then issues a `Connect` command, there will be contention for the stream head, since the stream head is locked until the IOCTL command completes. In blocking mode, the `Connect` command will be queued to run when the IOCTL completes. In non-blocking mode, the Connect will fail with a kEAGAINErr.

However, commands that read data from the endpoint (like `Rcv`, `RcvUData`, etc.) will only block waiting for data in synchronous/blocking mode. All other modes will return a kOTNoDataErr error if no data is available, or a kOTEAGAINErr if the command cannot be completed without blocking.

The blocking/non-blocking status of the provider also governs what happens when the provider is closed. In non-blocking mode, closing the provider will flush all outgoing commands in the stream, and immediately close the provider. In blocking-mode, the stream will be given up to 15 seconds per module to allow outgoing commands to be processed, then the stream will be closed.

At creation, all providers are set by default to non-blocking mode.

This call will always succeed.

**RESULT CODES**

`None`

**SEE ALSO**

`SetNonBlocking, IsNonBlocking, SetSynchronous, SetAsynchronous IsSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper, CloseProvider`






# SetNonBlocking

**FUNCTION**

`SetNonBlocking` Do not block (wait) for completion of outstanding I/O requests on a provider.

**C INTERFACE**

```
OSStatus    OTSetNonBlocking(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::SetNonBlocking();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

Like `SetBlocking`, `SetNonBlocking` governs how I/O requests proceed over the specified provider. See the description for `SetBlocking` for details.

At creation, all providers are set by default to non-blocking mode.

This call will always succeed.

**RESULT CODES**

`None`

**SEE ALSO**

`SetBlocking, IsNonBlocking, SetSynchronous, SetAsynchronous IsSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper`

# IsNonBlocking

**FUNCTION**

`IsNonBlocking` Return the current blocking status of a provider.

**C INTERFACE**

```
Boolean     OTIsNonBlocking(ProviderRef ref);
```

**C++ INTERFACE**

```
Boolean     TProvider::IsNonBlocking();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`IsNonBlocking` returns true if the specified provider is non-blocking, or returns false if the provider is blocking.

**RESULT CODES**

`none`

**SEE ALSO**

`SetNonBlocking, SetBlocking, SetSynchronous, SetAsynchronous IsSynchronous, OpenEndpoint, AsyncOpenEndpoint, OpenMapper, AsyncOpenMapper`






# AckSends

**FUNCTION**

`AckSends` If the provider is no longer referencing the client's data, notify the client each time a Send operation is completed on the provider.

**C INTERFACE**

```
OSStatus    OTAckSends(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::AckSends();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

If `AckSends` completes successfully, the client's event handler is called each time an operation that sends data is completed on the specified provider. `AckSends` provides a way for the client to request that Open Transport not copy the client data, and provides for an acknowledgment to be delivered to the client via the notification routine when the provider is finished with the data.

The client will receive a `T_MEMORYRELEASED` event in the notification routine. The `cookie` parameter will point to the buffer that was sent and the `result` parameter will be set to its length. Until the `T_MEMORYRELEASED` event is received, the client should refrain from changing the contents of the buffer, or the results will be unpredictable.

**NOTE**: `AckSends` will return a `kOTAccessErr` if a notifier is not installed on the provider. It will return a kOTStateChangeErr if a write-type operation is currently outstanding on the provider (Snd, SndUData, SndUReply, SndURequest, SndReply, or SndRequest).

In addition, using AckSends makes it dangerous to close a provider until all outstanding calls on the provider are completed. Otherwise, the application might quit, leaving Open Transport using memory that is no longer valid, with unpredictable results.

The endpoint functions that initiate Send operations are `Snd`, `SndRequest`, `SndReply`, `SndURequest`, `SndUReply`, `SndUData`, and, if the call has associated data, `Connect` and `Accept`.

By default, the Open Transport Library does not acknowledge the completion of Send operations.

**WARNING**: Do **NOT** wait for a `T_MEMORYRELEASED` event from a previous Send operation to trigger more sends. When a `T_MEMORYRELEASED` event occurs depends on how the underlying provider is implemented. It may hold on to memory until the next send occurs, or have some other functionality which causes it to delay releasing memory.

**RESULT CODES**

`kOTStateChangeErr`

**SEE ALSO**

DontAckSends, IsAckingSends, CloseProvider






# DontAckSends

**FUNCTION**

| | |
|---|---|
| DontAckSends | Do not notify the client when function calls that send data are completed on this provider. |

**C INTERFACE**

```
OSStatus    OTDontAckSends(ProviderRef ref);
```

**C++ INTERFACE**

```
OSStatus    TProvider::DontAckSends();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |

If DontAckSends completes successfully, the client's notification routine is no longer called when Send operations are completed on the specified endpoint. Instead, when the client calls a function that sends client data, the Open Transport Library copies the data into internal buffers, and the function returns immediately to the client. Thus, the client can immediately reuse the data area without affecting the Send operation, even if it is still in progress.

The endpoint functions that send data are Snd, SndRequest, SndReply, SndURequest, SndUReply, SndUData, and, only if the call has associated data, Connect and Accept.

**NOTE**: DontAckSends will return a kOTStateChangeErr if a write-type operation is currently outstanding on the provider (Snd, SndUData, SndUReply, SndURequest, SndReply, or SndRequest).

By default, the Open Transport Library does not acknowledge the completion of Send operations.

**RESULT CODES**

kOTStateChangeErr

**SEE ALSO**

AckSends, IsAckingSends

# IsAckingSends

**FUNCTION**

`IsAckingSends` Return the current status of acking sends.

**C INTERFACE**

```
Boolean     OTIsAckingSends(ProviderRef ref);
```

**C++ INTERFACE**

```
Boolean     TProvider::IsAckingSends();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`IsAckingSends` returns true if the specified provider is acking sends, or returns false if the provider is not acking Sends.

**RESULT CODES**

`none`

**SEE ALSO**

`AckSends, DontAckSends`






# Ioctl

**FUNCTION**

`Ioctl` Send a provider-specific command.

**C INTERFACE**

```
OTResult    OTIoctl(ProviderRef ref, UInt32 cmd, void* data);
```

**C++ INTERFACE**

```
OTResult    TProvider::Ioctl(UInt32 cmd, void* data);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref (C only)` | x | / |
| `cmd` | x | / |
| `data` | x | (?) |

The `Ioctl` function allows commands to be sent to a provider that are provider specific. The `cmd` and `data` parameters are documented with the provider that implements the commands. In some cases the `data` parameter is interpreted as a pointer, and sometimes it is just interpreted as a bit pattern.

In all cases, the return value of the `Ioctl` function (or the `result` parameter of a `kStreamIoctlEvent`, which is what you receive if the Ioctl is executed asynchronously) is zero or a positive number if no error occurred, and is one of the negative result codes if an error occurred.

**RESULT CODES**

Return codes are provider and `cmd` parameter specific

**SEE ALSO**

# CancelSynchronousCalls

**FUNCTION**

`CancelSynchronousCalls` Cancel any pending synchronous call on a provider.

**C INTERFACE**

```
OSStatus    OTCancelSynchronousCalls(ProviderRef ref, OSStatus err);
```

**C++ INTERFACE**

```
OSStatus    TProvider::CancelSynchronousCalls(OSStatus err);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `err` | / | x |

`CancelSynchronousCalls` cancels, or breaks out of, a synchronous call. The error code returned to the synchronous caller will be the "`err`" value supplied (typically `kOTCanceledErr`). A client will typically issue this call as the result of some timer process. However, be aware that making this call may cause a provider to be unusable. Typically, once this call is made, the only thing you can do with the provider is close it.

**RESULT CODES**

no specific result codes

**SEE ALSO**

`SetSynchronous, IsSynchronous`






# TransferOwnership

**FUNCTION**

`TransferOwnership` Transfer the ownership of a provider to the current Open Transport client.

**C INTERFACE**

```
ProviderRef OTTransferProviderOwnership(ProviderRef ref, OSStatus*
            errPtr);
```

**C++ INTERFACE**

```
ProviderRef TProvider::TransferOwnership(OSStatus* errPtr);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `errPtr` | x | (x) |

`TransferOwnership` transfers the ownership of a provider from whoever created the provider to the current Open Transport client.

It is necessary to use this call whenever some other entity (typically a shared library) opened an endpoint on behalf of the client. This is especially important if the entity that opened the endpoint is in a different architecture (PowerPC native vs. 68K emulated) than the client that will be using the provider.

Under Open Transport, a provider allocates a small amount of memory (currently 4 bytes) from the client for the endpoint. In addition, Open Transport automatically cleans up behind clients that call `CloseOpenTransport`. If a shared library creates a provider on your behalf, and that shared library subsequently unloads while you are still using the provider, two bad things happen. The memory for the provider is no longer valid, and the provider is closed out from underneath you. Using the `TransferOwnership` API, ownership is changed to the calling client. This means a new "ProviderRef" will be allocated in the calling client's memory pool, and the provider will be marked as belonging to the calling client.

As long as the entity that created the provider remains loaded, and is in the same architecture as the client using the provider, no damage will be done by not making this call. However, if the provider was created under a different architecture than the client using the provider, attempting to close the provider will cause a crash. It is vital that if you do not use the TransferOwnership API the provider be closed under the same architecture that opened the provider. In addition, when installing a notifier into the provider, Open Transport always assumes that the `OTNotifyProcPtr` is in the same architecture as the call is being made, so after transferring ownership, you may want to remove any notifier that is installed and install your own, unless your architecture is such that a cross-architecture notifier is what you want.

**RESULT CODES**

`kENOMEMErr`

# About Endpoints

The next concept to understand is the endpoint. This section describes what endpoints are, explains how they are created, and lists the kinds of endpoints.

An endpoint is a type of provider which specifies the communication path between a transport user (the client) and a specific transport provider. A transport provider is the actual code that provides the service of moving data. All requests to the transport provider must pass through an endpoint. A single endpoint can support only one established transport connection at a time; however, one transport provider may have several endpoints as clients.

A client creates an endpoint by calling the function `OTOpenEndpoint`, which returns an identifier to the newly created endpoint (called an `EndpointRef`). The client uses this identifier when making function calls to the endpoint. For example, a client must pass the identifier as the first parameter to any XTI-compatible functions of the Open Transport Library, thereby specifying which endpoint is to service the function call. When finished with an endpoint, a client calls the `OTCloseProvider` function, which destroys the endpoint and releases any associated system resources, such as memory.

## Endpoint Types

The XTI specification defines two kinds of data-stream endpoints:

- Connectionless datagram
- Connection-oriented stream

Open Transport supports these two kinds of data-stream endpoints; it also extends XTI by adding two kinds of transaction endpoints:

- Connectionless transaction
- Connection-oriented transaction

A connectionless-datagram endpoint provides datagram service. The UDP protocol of TCP/IP and the AppleTalk DDP protocol are examples of connectionless-datagram protocols.

A connection-oriented stream endpoint provides data transfer between two particular endpoints. A connection must be established between the endpoints before data transfer can take place. The TCP and AppleTalk ADSP protocols are connection-oriented streams. Each of these protocols is running over a connectionless datagram (TCP runs on top of IP; ADSP runs on top of DDP). Serial connections and modem connections are also connection-oriented streams. Note that a protocol such as PPP (Point-to-Point Protocol), which is a connectionless datagram, normally runs on top of serial connection, which is a connection-oriented stream.

A transaction protocol matches incoming responses with their corresponding requests. An example of a connectionless transaction protocol is the ATP protocol in AppleTalk; the AppleTalk ASP protocol is an example of a connection-oriented transaction protocol.

Both connection-oriented stream endpoints and connection-oriented transaction endpoints have a variant that supports an orderly connection tear-down feature called orderly release.






## Endpoint States

All Open Transport endpoints maintain a current state, which can be determined by the endpoint's client. Many Open Transport functions may be issued only when the endpoint is in a certain state. For example, it is not legal to send data on a connection-oriented endpoint unless there is an active connection. If a function call is issued when the endpoint state is not appropriate, an error will result (`kOTOutStateErr`).

The following table lists and briefly describes the endpoint states. For more information about these states, refer to Appendix B.

| State | Meaning |
|---|---|
| T_UNINIT | This endpoint has been closed and destroyed - probably by the computer going into sleep mode. |
| T_UNBND | This endpoint is initialized, but has not yet been bound to a local protocol address. |
| T_IDLE | This endpoint has been bound to a local protocol address and is ready for use. If it is a connectionless endpoint, then data may be sent and received. Connection-oriented endpoints may now issue `OTConnect` or `OTListen` requests. |
| T_INCON | This connection-oriented endpoint has received a connection request, but the client has not yet accepted or rejected the request. |
| T_DATAXFER | This connection-oriented endpoint has a connection established; the endpoint can now send and receive data. |
| T_OUTCON | The client has initiated a connection request on a connection-oriented endpoint, and the connection has not yet been established. |
| T_INREL | This connection-oriented endpoint has received an incoming request for an orderly disconnect, but the client has not yet acknowledged the release. |
| T_OUTREL | The client has initiated an orderly disconnect, but the remote endpoint has not yet acknowledged the request. |

The successful completion of some Open Transport functions causes an endpoint to change state. Also, asynchronous events can cause the endpoint to change state. Because certain functions are valid only for certain states, the order in which a client is allowed to make function calls is restricted. For more information, refer to the sections "Connectionless Endpoints" and "Connection-Oriented Endpoints."

Table 1-1 shows each of the Open Transport functions that can cause an endpoint's state to change. The center column shows the state following successful completion of the function, and the right column shows the state if the function is completed with an error. (For descriptions of these functions, refer to the section "Endpoint Functions.")

**Table 1-1.** Open Transport functions that can change an endpoint's state

| Function | Prior State | No Error | If Error |
|---|---|---|---|
| OpenEndpoint | N/A | T_UNBND | N/A |
| CloseProvider | Any | T_UNINIT | N/A |
| Bind | T_UNBND | T_IDLE | T_UNBND |
| Unbind | T_DATAXFER, T_IDLE | T_UNBND | Prior State |
| Connect | T_IDLE | T_OUTCON | T_IDLE |
| SndOrderlyDisconnect | T_DATAXFER | T_OUTREL | T_DATAXFER |
| SndOrderlyDisconnect | T_INREL | T_IDLE | T_INREL |
| RcvOrderlyDisconnect | T_DATA_FER | T_INREL | T_DATAXFER |
| RcvOrderlyDisconnect | T_OUTREL | T_IDLE | T_OUTREL |
| SndDisconnect | T_OUTCON, T_INCON, T_DATAXFER, T_OUTREL, T_INREL | T_IDLE | Prior State |
| RcvConnect | T_OUTCON | T_DATAXFER | T_IDLE |
| Accept | T_INCON | T_DATAXFER | T_IDLE or T_INCON |

Table 1-2 shows each of the Open Transport asynchronous events that can cause an endpoint's state to change. (For more information about these events, refer to Appendix C.)

**Table 1-2.** Open Transport asynchronous events that can change an endpoint's state

| Event | New State |
|---|---|
| T_LISTEN | T_INCON |
| T_CONNECT | T_DATAXFER |
| T_PASSCON | T_DATAXFER |
| T_DISCONNECT | T_IDLE |
| T_ORDREL | T_INREL |

# Connectionless Endpoints

Endpoints are either connectionless or connection-oriented. This section gives an overview of how a client uses a connectionless endpoint. The information in this section applies to both kinds of connectionless endpoints: connectionless datagram and connectionless transaction.

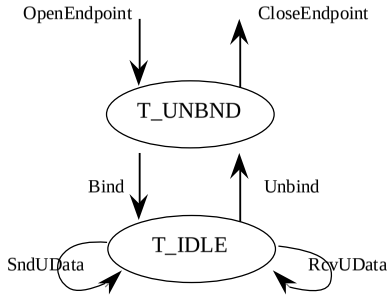
When a client creates an endpoint, by calling `OTOpenEndpoint`, the endpoint is inthe state T_UNBND. The client must then use the `Bind` function to assign a local protocol address to the endpoint. A client can let the endpoint assign the protocol address, or the client can request a specific protocol address. Only one connectionless endpoint can be bound to a single protocol address.

After binding, the endpoint's state changes to T_IDLE. The endpoint can then receive any incoming unit data or unit requests. The client can read incoming data or requests by calling the function `RcvUData` or `RcvURequest`, depending on the type of endpoint (connectionless datagram or connectionless transaction, respectively). The client can send data or requests by calling the function `SndUData` or `SndURequest`.






To return the endpoint to the T_UNBND state, the client uses the `Unbind` function. `Unbind` causes all protocol activity on the endpoint to stop. The client can then bind the endpoint again and reuse it, or can destroy it by calling the `CloseProvider` function.




**Simplified State Diagram for a Connectionless Endpoint**

# Connection-Oriented Endpoints

This section gives an overview of how a client uses a connection-oriented endpoint. The information in this section applies to both kinds of connection-oriented endpoints: connection-oriented stream and connection-oriented transaction.

When a client creates an endpoint, by calling `OpenEndpoint`, the endpoint is inthe state T_UNBND. The client must then use the `Bind` function to assign a local protocol address to the endpoint. A client can let the endpoint assign the protocol address, or the client can request a specific protocol address.

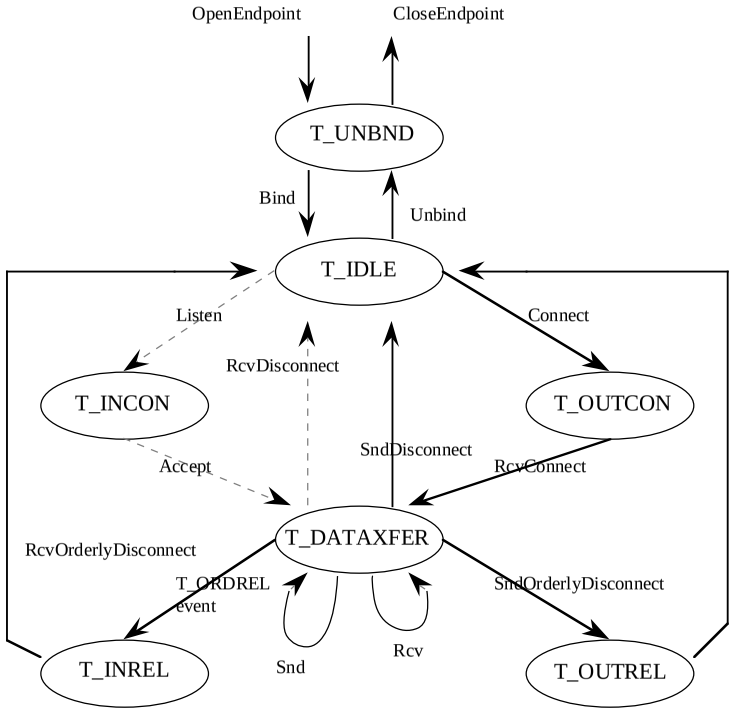
The client must use the `Bind` function to assign a local protocol address to the endpoint. For example, in the case of an AppleTalk ADSP endpoint, binding means assigning a DDP socket number to the endpoint. For all endpoints, a client may allow the endpoint to assign the protocol address, or the client may ask for a specific protocol address. For connection-oriented endpoints, it is possible for several endpoints to be bound to the same protocol address. However, only one of these endpoints may be designated to receive incoming connection requests. Because connections can be accepted on a different endpoint from the one that received the connection request, it is possible to have many open sessions (on different endpoints) sharing the same protocol address.

After binding, the endpoint's state changes to T_IDLE. The endpoint is then ready to receive any incoming connection requests, or to initiating an outgoing connection request.

The following figure shows the progression of states for a connection-oriented endpoint and shows the functions that cause the endpoint to change its state. For more information about how connection-oriented endpoints work, refer to the section "Using Connections."






**Simplified State Diagram for a Connection-Oriented Endpoint**

# Polling for Asynchronous Events

Clients can poll endpoints for asynchronous events but not for completion events. The client can read the most important pending asynchronous event by calling the function `OTLook`. The event is cleared automatically as the client performs some function call that consumes the event. For example, the T_LISTEN event that signals an incoming connection request is cleared when the client executes the Listen() function. The T_GODATA and T_GOEXDATA events are cleared by the `OTLook` function. For more information, refer to the description of the `Look` function.

# Using Asynchronous Mode

Open Transport clients must call most networking and communications functions asynchronously, because the Macintosh Operating System has no built-in threads facility. Although clients can call some of these functions synchronously, doing so generally results in a poor user experience, as the user's system can do nothing else while a call is in progress.

A client can make Open Transport calls asynchronously by first putting the endpoint into asynchronous mode with the `SetAsynchronous` function. In asynchronous mode, many functions return before performing their task, and send an event to the client when the function has completed. Some of the functions are inherently synchronous and have no corresponding completion event. For example, the `GetEndpointState` function always returns the current endpoint state right away. If a function behaves differently in asynchronous mode, that behavior is described in the description of the function.

A client that uses an endpoint in asynchronous mode must be prepared for the notifier to be called signaling a function completion **before** the function actually returns to the client. For example, a client could make an `OTSndUReply` to send a reply using an endpoint in asynchronous mode, and the client's notification routine may be called with the `T_REPLYCOMPLETE` event even before the `OTSndUReply` function returns.

A function that is asynchronous can return an error immediately, or it can return an error to the notification function. However, only certain errors will be returned immediately. These are:

| | |
|---|---|
| kOTStateChangeErr | The requested command cannot be executed because an incompatible command is still outstanding. |
| kOTOutStateErr | The requested command cannot be executed because the endpoint is not in the correct state for the command. |
| kENOMEMErr | There is not enough memory available to begin command execution. |
| kEINVALErr | The arguments to the command were invalid. |
| kOTLookErr | An event has occurred which requires that it be dealt with before your command is executed. Use the OTLook function to determine which event occurred. |
| kOTBadAddressErr | An address parameter to the function was invalid. |
| kOTBadOptionErr | An option parameter to the function was invalid. |
| kOTBadDataErr | A data parameter to the function was invalid. |
| kOTBadFlagErr | The OTFlags parameter to the function was invalid. |
| kOTBadSequenceErr | The sequence parameter to the function was invalid. |
| kOTNoDataErr | The command requests incoming data and none is available. |
| kOTBadQLenErr | A Listen call was made on an endpoint that was bound with a qlen parameter of zero. |
| kOTNotSupportedErr | The endpoint does not support the function call that was made. |
| kOTFlowErr | The command requested data to be sent, but the endpoint is currently flow-controlled and cannot send data. |






Here is a list of the Open Transport functions that operate differently in synchronous and asynchronous mode. Next to the name of each such function is the event issued when the function is completed:

| | |
|---|---|
| `OptionManagement` | T_OPTIONMGMTCOMPLETE |
| `Bind` | T_BINDCOMPLETE |
| `Unbind` | T_UNBINDCOMPLETE |
| `Accept` | T_ACCEPTCOMPLETE |
| `SndRequest` | T_REQUESTCOMPLETE |
| `SndReply` | T_REPLYCOMPLETE |
| `SndURequest` | T_REQUESTCOMPLETE |
| `SndUReply` | T_REPLYCOMPLETE |
| `Disconnect` | T_DISCONNECTCOMPLETE |
| `GetProtAddress` | T_GETPROTADDRCOMPLETE |
| `ResolveAddress` | T_RESOLVEADDRCOMPLETE |

**Note:** Open Transport asynchronous functions operate differently from the way that Macintosh device driver asynchronous functions do. Macintosh device drivers almost always return "no error" for function calls made asynchronously, and all result codes are passed to the completion routine. (The `_Control` trap on the Macintosh returns an immediate error in asynchronous mode only if there is an invalid control code in the parameter block or the driver refNum is invalid.) All Open Transport asynchronous functions return a result code.

## Multiple Outstanding Asynchronous Calls

For many of the Open Transport functions, it is possible to have several concurrent outstanding instances of a call. For example, a client can issue several `OTResolveAddress` calls on the asynchronous endpoint in a row. Similarly, a client can have several `OTSndURequest` calls pending on the same transaction oriented endpoint.

When Open Transport calls the client notification routine to tell the client that a function has completed, the `cookie` parameter to the notification routine is used to help distinguish which instance of a call completed. For example, the `OTResolveAddress` function takes a pointer to a `TCall` structure containing the address to resolve and a pointer to a `TCall` to hold the resolved address. When the `OTResolveAddress` function completes, the address of the second `TCall` is passed in the `cookie` parameter to the notification routine along with a T_RESOLVEADDRCOMPLETE event code. The meaning of the `cookie` is described in the event code section of this document.

# Handling Events for endpoints

Event handling for endpoints is the same as that described for providers. There is one cautionary note to be aware of when dealing with the T_DATA, T_EXDATA, and T_REQUEST events. These events are used by the Open Transport Library to signal the arrival of incoming data or an incoming transaction request. For efficiency, Open Transport notifies the client only **once** that incoming data has arrived. To read all the data, the client must repeatedly issue the consuming Open Transport function (`Rcv`, `RcvUData`, `RcvRequest`, or `RcvURequest`) until the function returns with a `kOTNoDataErr` error. The client does not have to issue these calls in the notification routine itself, but until the client makes the consuming calls and receives a `kOTNoDataErr` error, another T_DATA, T_EXDATA, or T_REQUEST event will not be issued. A client should also be prepared for being notified that data is available, but then receiving a `kOTNoDataErr` error when trying to read the data.

One exception to this rule occurs when dealing with transaction protocols. When the client gets a `T_REPLY` event , OTRcvUReply is called until a kOTNoDataErr is returned. If this is deferred from the notification function to the foreground, the following sequence can occur: While the client is busy reading replies in the foreground, a request arrives. This will cause a T_REQUEST event to be generated. If the foreground client was calling OTRcvUReply at this point in time, a kOTLookErr will be generated rather than a kOTNoDataErr. In this case (and the converse case for T_REQUEST events), another T_REPLY event will be generated when a new reply arrives.

If we look at this operationally, the transport provider has a queue of data/commands to deliver to the client. If the queue is empty when the data/command arrives, a notification is delivered to the client. If the queue is not empty, then no notification is delivered to the client at the time the data/command is queued. Instead, whenever the client reads the data/command at the head of the queue, Open Transport peeks at the next element of the queue, if it exists. If this next element of the queue is of the same type as what was at the head of the queue, no event is generated. If there is a difference, a new event is delivered to the client. This new event is typically delivered to the client just prior to returning from the function which removed the head element of the queue.

# Open Transport Data Structures

This section describes the general data structures and typedefs used by Open Transport clients. Routine-specific structures are covered in the descriptions of the routines that use them.

Open Transport uses many typedefs to distinguish the different values used in it's routines. This first list are typedefs that are used solely for parameter passing. The use of these typedefs is to account for differences in the way that "C" compilers pass parameters:

| | |
|---|---|
| uchar_p | Used when passing an unsigned char parameter. |
| ushort_p | Used when passing an unsigned short parameter. |
| short_p | Used when passing a short parameter. |
| char_p | Used when passing a char parameter |
| boolean_p | Used when passing a Boolean parameter. |

These typedefs are all set to an equivalent data type that is 4 bytes long.

The next set of typedefs are used to give better information about the parameter than just the size and signed / unsigned characteristics of the value:

| | |
|---|---|
| OTRelease | An internal typedef used by Open Transport |
| OTTimeout | A timeout value, specified in milliseconds. |
| OTBand | A "band" number for use when using the "raw" Streams APIs |
| OTSequence | A sequence value used for matching transactions and connection requests. |
| OTNameID | An ID returned by a mapper that uniquely identifies a registered name |
| OTReason | A reason code returned from the OTRcvDisconnect function |
| OTQLen | A qlen value passed to the OTBind function |
| OTClient | A value that uniquely identifies an Open Transport client |
| OTClientName | A typedef for the name of an Open Transport client |
| OTOpenFlags | A value for the flags passed to the various Open Transport "open" routines |
| OTUnixErr | A positive error code corresponding to one of the Unix error codes |
| OTXTIErr | A positive error code corresponding to one of the XTI error codes |
| OSStatus | A negative error code. An OSStatus never has a signed value larger than 0. |
| OTResult | A value that holds either a result code or an error code. If the value is negative, an error occurred and the value is the error code. If the value is 0 or positive, no error occured and the value's interpretation depends on the function that was called. |
| OTAddressType | A value describing an address type used in Open Transport functions which require a protocol address |






| | |
|---|---|
| OTStructType | A value describing the various structures used by Open Transport, used by the OTAlloc procedure. |
| OTFlags | A value describing the flags used for sending and receiving data |
| OTEventCode | A value that indicates an Open Transport event. |
| OTNotifyProcPtr | A typedef for the Open Transport standard "notifier" or call-back procedure |
| OTXTILevel | A value that indicates the XTI level number of a protocol, used in OTOptionManagement calls |
| OTXTIName | A value that indicates the name of a protocol option, used in OTOptionManagement calls. |
| OTPortRef | A value that holds a unique identifier for an Open Transport port / driver |
| OTProcessProcPtr | A typedef for a call-back procedure for the Open Transport scheduling functions. |
| OTTimeStamp | A value that holds an Open Transport time value. |

Open Transport clients use several general data structures:

| | |
|---|---|
| `TNetbuf` | Refers to variable-length fields such as a protocol address, protocol options, or data. (XTI defines an identical structure, `netbuf`.) |
| OTData | A structure used to pass non-contiguous data to Open Transport functions |
| `TEndpointInfo` | Contains information about an endpoint such as its maximum data size, maximum protocol-address length, and type of service provided. |

## TNetbuf Structure

The C definition of a `TNetbuf` structure is:

```
struct TNetbuf
{
    UInt32 maxlen;
    UInt32 len;
    UInt8* buf;
};
```

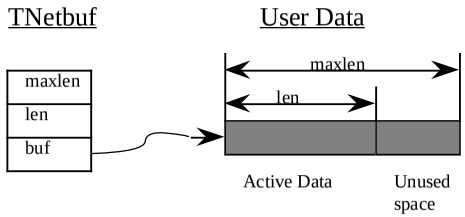
The meaning of the fields in a `TNetbuf` structure is as follows:

| | |
|---|---|
| `maxlen` | The maximum size of the buffer. The client sets this field before passing the `TNetbuf` to an Open Transport function, if the endpoint will return information in the buffer. If the endpoint has more information to return than fits in the `TNetbuf`, a `kOTBufferOverflowErr` error is usually returned. |
| `len` | The amount of valid data in the buffer. In Open Transport functions where the client passes data to the endpoint, the client is responsible for setting this field. (The `maxlen` field is usually ignored in this case.) In Open Transport functions that fill a `TNetbuf` supplied by the user, the endpoint sets this field in the `TNetbuf` to indicate how many bytes of data were actually returned. |

`buf` A pointer to the actual data buffer.

**IMPORTANT:** `TNetbuf` is just a data structure that *refers* to a data buffer; the `TNetbuf` does not actually `contain` the data. The client is responsible for ensuring that the `buf` field points to a data area.

The following diagram shows a `TNetbuf` structure and the data area it to which it refers.




**A `TNetbuf` structure**

# OTData Structure

The `OTData` structure may be used to send non-contiguous data to certain Open Transport functions. It is only supported in the functions Snd, SndUData, SndURequest, SndUReply, SndRequest, and SndReply.

**WARNING**: This is an Apple extension, and using it will cause your program not to work if ported to other XTI/STREAMS environments.

The structure is used by `Snd` by passing a pointer to the `OTData` as the buffer parameter, and using the constant `kNetbufDataIsOTData` as the length. All other functions pass the `OTData` as the `buf` field of a `TNetbuf`, setting the `len` field to `kNetbufDataIsOTData`. This is only valid for the data `TNetbuf`. Address and option `TNetbufs` are not allowed to use `OTData` structures.

The C definition of an `OTData` structure is:

```
struct OTData
{
    OTData*    fNext;
    void*      fData;
    size_t     fLen;
};
```

The meaning of the fields in an `OTData` structure is as follows:

`fNext` A pointer to the next OTData in the list. NULL for the last element in the list of data.

`fData` A pointer to the actual data for this fragment.

`fLen` The number of bytes in this fragment.






# TEndpointInfo Structure

The `TEndpointInfo` structure contains information about an endpoint. The declaration is:

```
struct TEndpointInfo
{
    SInt32    addr;
    SInt32    options;
    SInt32    tsdu;
    SInt32    etsdu;
    SInt32    connect;
    SInt32    discon;
    UInt32    servtype;
    UInt32    flags;
};
```

The meaning of the fields in `TEndpointInfo` is as follows:

`addr` A value greater than zero indicates the maximum size of a protocol address. A value of T_INVALID indicates that the endpoint does not provide the client access to the address.

`options` A value greater than zero indicates the maximum number of bytes needed to hold the protocol-specific options supported by the endpoint. A value of T_INVALID indicates that the endpoint does not support client-settable options.

`tsdu` The value in this field has one of two meanings, depending upon the type of endpoint that is being interrogated. For streams (both connectionless and connection-oriented), this field has the following meaning:

A value greater than zero indicates the maximum size of a transport service data unit (TSDU). A value of zero indicates that the endpoint does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection. A value of T_INFINITE specifies that there is no limit to the size of a TSDU. A value of T_INVALID indicates that the endpoint does not support normal data.

For transactions (both connectionless and connection-oriented), this field has the following meaning:

This field must be greater than zero; it indicates the maximum size of a response supported by the endpoint.

`etsdu` The value in this field has one of two meanings, depending upon the type of endpoint that is being interrogated. For streams (both connectionless and connection-oriented), this field has the following meaning:

A value greater than zero indicates the maximum size of an expedited transport service data unit (ETSDU). A value of zero indicates that the endpoint does not support the concept of ETSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection. A value of T_INFINITE specifies that there is no limit to the size of an ETSDU. A value of T_INVALID indicates that the endpoint does not support expedited data.

**Note:** The semantics of expedited data may be different for different kinds of endpoints.

For transactions (both connectionless and connection-oriented), this field has the following meaning:

This field must be greater than zero; it indicates the maximum size of a request supported by the endpoint.

`connect` A value greater than zero indicates the maximum amount of data (in bytes) that may be associated with connection establishment functions. A value of T_INVALID indicates that the endpoint does not allow data to be sent with connection establishment.

`discon` A value greater than zero indicates the maximum amount of data that may be associated with the `SndDisconnect` and `RcvDisconnect` functions. A value of T_INVALID indicates that the endpoint does not allow data to be sent with the abortive release functions.

`servtype` This field specifies the service type provided by the endpoint.

T_COTS indicates that the endpoint supports connection-oriented service but does not support orderly release, which is optional.

T_COTS_ORD indicates that the endpoint supports a connection-oriented service with the optional orderly release.

T_CLTS indicates that the endpoint supports a connectionless service. For this service, the `etsdu`, `connect`, and `discon` fields will all be T_INVALID.

T_TRANS indicates that the endpoint supports a connection-oriented transaction service, but does not support orderly release, which is optional.

T_TRANS_ORD indicates that the endpoint supports a connection-oriented transaction service with optional orderly release.

T_TRANS_CLT indicates that the endpoint supports a connectionless transaction service.

`flags` This is a bit field used to specify other information about the endpoint. If the T_SENDZERO bit is set in `flags`, this indicates that the underlying transport provider supports the sending of zero-length TSDUs.

Clients use the sizes provided in this structure to determine how large any required buffers must be to hold each piece of information. A client should never make assumptions about the size of data structures just because the client knows which endpoint it is using.

# Opening and Closing Endpoints

Before calling any other endpoint functions, a client must first create the endpoint by calling the function `OTOpenEndpoint`. This function takes one parameter, an OTConfiguration structure, which can be created using a smple ASCII string that gives the name of the endpoint. Based on this OTConfiguration structure, the Open Transport Library creates the endpoint. Typically, creating an endpoint will require that the Open Transport Library make some assumptions. For example, opening an endpoint using the value OTCreateConfiguration(“ADSP”) will cause an ADSP endpoint to be created. But an ADSP endpoint, by itself, is of no use; it requires a complete underlying protocol stack. The implementation for opening most endpoints will contain heuristics to use some default configurations. In this case, the ADSP implementation of opening an endpoint creates a DDP module running over the user’s default AppleTalk port.

When finished using an endpoint, a client must call the `OTCloseProvider` function, which tears down the endpoint along with any lower-layer protocols associated with it.






`OTOpenEndpoint` is described on the following pages.

# OTOpenEndpoint

**FUNCTION**

`OpenEndpoint` Create an endpoint.

**C INTERFACE**

```
EndpointRef OTOpenEndpoint(OTConfiguration* config, OTOpenFlags oflag,
            TEndpointInfo* info, OSStatus* err)
```

**C++ INTERFACE**

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

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| config | x | / |
| oflag | x | / |
| info->addr | / | x |
| info->options | / | x |
| info->tsdu | / | x |
| info->etsdu | / | x |
| info->connect | / | x |
| info->discon | / | x |
| info->servtype | / | x |
| info->flags | / | x |
| err | / | x |

`OTOpenEndpoint` creates an endpoint based on the supplied information, and returns a value by which the created endpoint can be identified when calling other endpoint functions.

The endpoint will be opened in synchronous, non-blocking mode.

The `config` parameter is a pointer to an OTConfiguration structure. The client cannot create one of these structures manually, but instead must use the function OTCreateConfiguration:

```
pascal OTConfiguration* OTCreateConfiguration(char* path);
```

This function takes a string parameter (typically the endpoint name), creates an `OTConfiguration` structure and returns a pointer to it to the client. The client should pass this pointer to `OTOpenEndpoint`. The `OTOpenEndpoint` function will destroy the structure. An example of calling `OTOpenEndpoint` using this function is shown below:

```
TEndpoint info;
OSStatus err;
EndpointRef ep = OTOpenEndpoint(OTCreateConfiguration("ddp"), 0, &info, &err);
```

The parameter `oflag` is not currently used and should be set to zero.

`OTOpenEndpoint` also returns several default characteristics of the endpoint in the `info` parameter, which is of type `TEndpointInfo`. Clients use the sizes provided in this structure to determine how large any required buffers must be to hold each piece of information. A client should never make assumptions about the size of data structures just because the client knows which endpoint it is using. If the `info` parameter is `NULL`, `OTOpenEndpoint` returns no protocol information.






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

The output parameter `err` points to a result code.

**RESULT CODES**

`kOTBadFlagErr`

`kOTBadNameErr`

`kOTCanceledErr`

**SEE ALSO**

`OTAsyncOpenEndpoint`, `OTCloseProvider`, `OTCreateConfiguration`, `OTCloneConfiguration`

# AsyncOpenEndpoint

**FUNCTION**

`AsyncOpenEndpoint` Create an endpoint asynchronously.

**C INTERFACE**

```
OSStatus    OTAsyncOpenEndpoint(OTConfiguration* config, OTOpenFlags
            oflag, TEndpointInfo* info, OTNotifyProcPtr proc, void*
            contextPtr)
```

**C++ INTERFACE**

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

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `config` | x | / |
| `oflag` | x | / |
| `info->addr` | / | x |
| `info->options` | / | x |
| `info->tsdu` | / | x |
| `info->etsdu` | / | x |
| `info->connect` | / | x |
| `info->discon` | / | x |
| `info->servtype` | / | x |
| `info->flags` | / | x |
| `proc` | x | / |
| `contextPtr` | x | / |

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

The `config`, `oflag`, and `info` parameters have the same meaning as for `OTOpenEndpoint`.

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

The endpoint will be opened in asynchronous, non-blocking mode, and will already have a notification routine installed, which is the same notification routine used for the open. If you want a different notifier installed, use `OTRemoveNotifier` to remove the current one, and use `OTInstallNotifier` to install a new one.

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






**RESULT CODES**

`kOTBadFlagErr`
`kOTBadNameErr`
`kOTCanceledErr`

**SEE ALSO**

`OpenEndpoint, CloseProvider, OTCreateConfiguration, OTCloneConfiguration`

# Binding and Unbinding

Before an endpoint can be used, it must be bound to a protocol address. Binding assigns a local address to the endpoint. The client can request to bind to a particular address, or the client can let the endpoint pick its own address.

No data transfer can take place on an endpoint until it is bound. Once a connectionless endpoint is bound, it will be able to receive incoming data, and a client may send data through the endpoint. For connection-oriented endpoints, the endpoint is ready to receive incoming connection requests or make outgoing connection requests.

An endpoint can be bound and unbound multiple times without closing the endpoint in between.






# Bind

**FUNCTION**

`Bind` Bind an address to an endpoint.

**C INTERFACE**

```
OSStatus    OTBind(EndpointRef ref, TBind* req, TBind* ret);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Bind(TBind* req, TBind* ret);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->addr.maxlen` | / | / |
| `req->addr.len` | x >= 0 | / |
| `req->addr.buf` | (x) | / |
| `req->qlen` | x >= 0 | / |
| `ret->addr.maxlen` | x | / |
| `ret->addr.len` | / | x |
| `ret->addr.buf` | ? | (?) |
| `ret->qlen` | / | x >= 0 |

`Bind` associates a local protocol address with the endpoint specified by the parameter `ref`. Most endpoint functions complete successfully only if the endpoint you specify is currently bound. For instance, a connectionless endpoint can send and receive data only if the endpoint is bound. Similarly, a connection-oriented endpoint can enqueue incoming connect indications, and its client can initiate a connection, only if the endpoint is bound.

The parameter `req` is a `TBind` structure, which has the following fields:

```
struct TNetbuf addr;
OTQLen qlen;
```

The parameter `req` is used to request the address given in the `req.addr` field of the `TBind` structure. Some endpoints treat the requested address as a suggestion or hint; the actual address that they bind to an endpoint may differ from the requested address. When the `Bind` function returns, the `TBind` structure to which the `ret` parameter points contains the address actually bound to the endpoint. The `ret->addr.len` field will contain the length of the address. If the `ret->addr.maxlen` field indicates the `TNetbuf` is not large enough to contain the address, an error (`kOTBufferOverflowErr`) will result.

The qlen field has meaning only when initializing a connection-oriented service. It specifies the number of outstanding connect indications that the endpoint should support. A outstanding connect indication is one that has been passed to the client but which has been neither accepted (via Accept) nor rejected (via SndDisconnect). A value of qlen greater than zero is meaningful only when Bind is issued by a passive client that expects other clients to connect to it. The value of qlen will be negotiated by the endpoint and may be changed if the endpoint cannot support the specified number of outstanding connection indications. For connection-oriented endpoints, this value of qlen will not be negotiated to zero from a requested value greater than zero. When the Bind call returns, the negotiated value is stored in the qlen field of the TBind structure specified by the ret parameter.

The ret parameter may be NULL if the client does not care what address the endpoint is bound to (or will use the OTGetProtAddress function to find out) and does not care about the negotiated value of qlen.

If the requested address is not available, an error will result (kOTAddressBusyErr). If no address is specified in the req parameter (req->addr.len is zero or req is NULL), the endpoint will assign an address. If the req parameter is NULL, the value for req->qlen is assumed to be zero. If the endpoint could not allocate an address, the function will fail with the kOTNoAddressErr error.

It is valid to set both req and ret to NULL for the same call.

A client must not bind multiple protocol address to a single endpoint. (However, some connection-oriented endpoints let a client bind multiple endpoints to a single protocol address. ) If a client binds more than one endpoint to the same protocol address, only one endpoint can be used to listen for connection indications for that address. In other words, only one Bind for a given protocol address may specify a qlen greater than zero.

If a client attempts to bind a protocol address to another endpoint using a qlen greater than zero, Bind will return the kOTAddressBusyErr error. When a client accepts a connection on the endpoint that is being used as the listening endpoint, the bound protocol address is busy for the duration of the connection, until an OTUnbind or OTCloseProvider call is issued. No other endpoints may be bound for listening on that same protocol address while the initial endpoint is active (either in the T_IDLE or T_DATAXFER states). This will prevent more than one endpoint bound to the same protocol address from accepting connect indications.

If the endpoint is connectionless, only one endpoint may be associated with a protocol address. If a client attempts to bind a second endpoint to an already bound protocol address, Bind will return the kOTAddressBusyErr error.

If the endpoint is in synchronous mode, the function will not return until the bind is complete.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the Bind function returns kOTNoError, a T_BINDCOMPLETE event will be issued when the bind completes. The result parameter will be kOTNoError if the bind completed successfully. Otherwise, it will contain a result code describing the reason that the bind failed. The cookie parameter passed to the notification routine holds the ret parameter.

If a notification routine has not been installed, the only way to determine that the bind has completed is to poll the GetEndpointState function. This function will return a kOTStateChangeErr until the bind completes. When the bind completes, the state will either be T_UNBND if the bind failed, or T_IDLE if it succeeded. If the bind failed, there is no mechanism for determining the result code that it failed with.

**Note:** In asynchronous mode, the T_BINDCOMPLETE event may be issued before the Bind function returns a kOTNoError result code.

An asynchronous Bind still in progress may be canceled by issuing the Unbind function.






> **CAUTION:** An endpoint may not allow an explicit binding of more than one endpoint to the same protocol address, although it allows more than one connection to be accepted for the same protocol address. To ensure portability, do not bind endpoints that are used as responding endpoints in a call to Accept, if the responding address is to be the same as the called address.

**VALID STATES**

T_UNBND

**RESULT CODES**

kOTAccessErr
kOTAddressBusyErr
kOTBadAddressErr
kOTBufferOverflowErr
kOTCanceledErr
kOTNoAddressErr

**SEE ALSO**

Unbind, GetEndpointState

# Unbind

**FUNCTION**

`Unbind` Return an endpoint to the unbound state.

**C INTERFACE**

```
OSStatus    OTUnbind(EndpointRef ref);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Unbind();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`Unbind` disables an endpoint previously bound by the `Bind` function. On completion of this call, no further data or events destined for this endpoint will be accepted. An endpoint which is disabled with the `Unbind` function can be enabled later by calling the `Bind` function.

If the endpoint is in synchronous mode, the function will wait until the unbind is completed.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the `Unbind` function returns `kOTNoError`, a `T_UNBINDCOMPLETE` event will be issued when the unbind completes. The result parameter will be `kOTNoError` if the unbind completed successfully. Otherwise, it will contain a result code describing the reason that the unbind failed (most often it is a kOTLookErr on connectionless endpoints, usually indicating that more data has arrived. Unfortunately, XTI defines that an Unbind will only succeed when there is no data available. The only recourse is to either read the data and try again, or close the endpoint). The `cookie` parameter passed to the notification routine has no meaning, and will be zero.

If a notification routine has not been installed, the only way to determine that the unbind has completed is to poll the `GetEndpointState` function. This function will return a `kOTStateChangeErr` until the unbind completes. When the unbind completes, the state will either be `T_IDLE` if the unbind failed, or `T_UNBND` if it succeeded. If the unbind failed, there is no mechanism for determining the result code that it failed with.

**Note:** In asynchronous mode, it is possible for the endpoint to issue the T_UNBINDCOMPLETE event before the `Unbind` function returns the `kOTNoError` result to the client.

**VALID STATES**

T_IDLE

**RESULT CODES**

`kOTBadReferenceErr`
`kOTBadSyncErr`
`kOTLookErr`
`kOTOutStateErr`






**SEE ALSO**

`Bind, GetEndpointState`

## Getting Information About an Endpoint

This section describes the functions that clients can use to get information about a particular endpoint.

# GetEndpointInfo

**FUNCTION**

`GetEndpointInfo` Return information about an endpoint.

**C INTERFACE**

```
OSStatus    OTGetEndpointInfo(EndpointRef ref, TEndpointInfo* info);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::GetEndpointInfo(TEndpointInfo* info);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `info->addr` | / | x |
| `info->options` | / | x |
| `info->tsdu` | / | x |
| `info->etsdu` | / | x |
| `info->connect` | / | x |
| `info->discon` | / | x |
| `info->servtype` | / | x |
| `info->flags` | / | x |

`GetEndpointInfo` gets information about the specified endpoint. It returns this information in the parameter `info`, which is of type `TEndpointInfo`.

If the `info` pointer argument to `GetEndpointInfo` is passed as a `NULL` pointer, `GetEndpointInfo` returns no protocol information.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the `GetEndpointInfo` function returns `kOTNoError`, a `T_GETINFOCOMPLETE` event will be issued when the function completes. The result parameter will be `kOTNoError` if the function completed successfully. Otherwise, it will contain a result code describing the reason that the function failed. The `cookie` parameter passed to the notification routine to indicate completion contains the value of the `info` parameter that was passed to the original function call.

If a notification routine has not been installed, it is not possible to determine when this command is completed.

**VALID STATES**

All

**RESULT CODES**

no specific result codes






**SEE ALSO**

GetEndpointState, Sync

# GetEndpointState

**FUNCTION**

`GetEndpointState` Return the current XTI state of an endpoint.

**C INTERFACE**

```
OTResult    OTGetEndpointState(EndpointRef ref);
```

**C++ INTERFACE**

```
OTResult    TEndpoint::GetEndpointState();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

On successful completion, `GetEndpointState` returns an integer value of zero or greater, indicating the XTI state of the specified endpoint. (Endpoint states are listed and described in Appendix B.) The only error returned by this function is `kOTStateChangeErr`, which indicates that the state of the endpoint is currently changing.

**VALID STATES**

All

**RESULT CODES**

`kOTStateChangeErr`

**SEE ALSO**

`OpenEndpoint, GetEndpointInfo, Sync`






# Look

**FUNCTION**

`Look` Return current event flags for an endpoint.

**C INTERFACE**

```
OTResult    OTLook(EndpointRef ref);
```

**C++ INTERFACE**

```
OTResult    TEndpoint::Look();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

On successful completion, `Look` returns an integer value of zero or greater, indicating the most important event pending on the specified endpoint. The only error returned is the `kOTStateChangeErr`.

Clients can use `Look` to poll for asynchronous events such as incoming data or a connection request.

Some functions return the `kOTLookErr` result code, indicating that the client should make the `Look` function call to determine why the original function aborted.

If successful, `Look` returns one of the event codes described in Appendix C (but not any of the event codes whose name ends in `COMPLETE`).

**VALID STATES**

All

**RESULT CODES**

`kOTStateChangeErr`

**SEE ALSO**

`InstallNotifier`

# GetProtAddress

**FUNCTION**

| | |
|---|---|
| `GetProtAddress` | Get the address to which the specified endpoint is bound. If the endpoint is connection-oriented and currently connected, also get the address to which it is connected. |

**C INTERFACE**

```
OSStatus    OTGetProtAddress(EndpointRef ref, TBind* boundAddr, TBind*
            peerAddr);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::GetProtAddress(TBind* boundAddr, TBind*
            peerAddr);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `boundAddr->addr.maxlen` | x | / |
| `boundAddr->addr.len` | / | x |
| `boundAddr->addr.buf` | ? | (?) |
| `boundAddr->qlen` | / | / |
| `peerAddr->addr.maxlen` | x | / |
| `peerAddr->addr.len` | / | x |
| `peerAddr->addr.buf` | ? | (?) |
| `peerAddr->qlen` | / | / |

`GetProtAddress` gets the local and remote protocol addresses currently associated with the endpoint. The client is responsible for initializing the `TNetbuf` structures in the `TBind` structure with buffers large enough to hold the addresses. (To find the length of the address, call the `GetEndpointInfo` function.)

The `TBind` structure has the following members:

```
struct TNetbuf addr;
OTQLen qlen;
```

The local address of the endpoint is returned in the `boundAddr` structure unless the endpoint is in the T_UNBND state. In the T_UNBND state, the `boundAddr->addr.len` field will be set to zero.

The remote address that the endpoint is connected to will be returned in the `peerAddr` structure. If the endpoint is not currently in the T_DATAXFER state or is not a connection-oriented endpoint, the `peerAddr->addr.len` field will be set to zero. The `peerAddr` pointer in the `GetProtAddress` function may be `NULL`.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the `GetProtAddress` function returns `kOTNoError`, a `T_GETPROTADDRCOMPLETE` event will be issued when the function completes. The result parameter will be `kOTNoError` if the function completed successfully. Otherwise, it will contain a result code describing the reason that the function failed. The `cookie` parameter passed to the notification routine to indicate completion contains the `peerAddr`, unless the `peerAddr` is `NULL`. If the `peerAddr` is `NULL`, the `cookie` contains the `boundAddr`.






If a notification routine has not been installed, it is not possible to determine when this command is completed.

**VALID STATES**

All

**RESULT CODES**

`kOTBufferOverflowErr`

**SEE ALSO**

Bind, Connect, Accept

# ResolveAddress

**FUNCTION**

`ResolveAddress` Resolve a protocol address.

**C INTERFACE**

```
OSStatus    OTResolveAddress(EndpointRef ref, TBind* req, TBind* ret,
            OTTimeout timeout);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::ResolveAddress(TBind* req, TBind* ret, OTTimeout
            timeout);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->addr.maxlen` | / | / |
| `req->addr.len` | x | / |
| `req->addr.buf` | (x) | / |
| `req->qlen` | / | / |
| `ret->addr.maxlen` | x | / |
| `ret->addr.len` | / | x |
| `ret->addr.buf` | ? | (?_ |
| `ret->qlen` | / | / |
| `timeout` | x | / |

`ResolveAddress` gets the lowest-layer protocol address that corresponds to the specified protocol address, within its protocol family. For example, `ResolveAddress` might convert a higher-layer address like "mynetwork.com" into a lower-layer address like "33.77".

The `TBind` structure has the following members:

```
struct TNetbuf addr;
OTQLen qlen;
```

If the endpoint is in asynchronous mode, a notification routine has been installed, and the `ResolveAddress` function returns `kOTNoError`, a `T_RESOLVEADDRCOMPLETE` event will be issued when the function completes. The result parameter will be `kOTNoError` if the function completed successfully. Otherwise, it will contain a result code describing the reason that the function failed. The `cookie` parameter passed to the notification routine to indicate completion contains the `ret` parameter that was passed to the `ResolveAddress` call.

The timeout parameter indicates the maximum time in milliseconds that you want to wait for address resolution to occur. This parameter is advisory only, and not all protocols will honor it.

**VALID STATES**

All except T_UNINIT






**RESULT CODES**

`kOTBufferOverflowErr`
`kOTNotSupportedErr`
`kOTOutStateErr`
`kOTStateChangeErr`
`kOTBadAddressErr`

**SEE ALSO**

None

# Sync

**FUNCTION**

| | |
|---|---|
| Sync | Insures that the transport provider and client are synchronized as to state and endpoint information. |

**C INTERFACE**

```
OTResult    OTSync(EndpointRef ref);
```

**C++ INTERFACE**

```
OTResult    TEndpoint::Sync();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |

On successful completion, Sync returns an integer value of zero or greater, corresponding to current information about the endpoint. On error, Sync returns a negative integer corresponding to a result code.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the Sync function returns kOTNoError, a T_SYNCCOMPLETE event will be issued when the function completes. The result parameter will be kOTNoError if the function completed successfully. Otherwise, it will contain a result code describing the reason that the function failed. The cookie parameter passed to the notification routine to indicate completion has no meaning.

If a notification routine has not been installed, it is not possible to determine when this command is completed.

It should not be necessary to call this function for Open Transport. It is provided for backward compatibility with the XTI interfaces.

**VALID STATES**

All

**RESULT CODES**

no specific result codes

**SEE ALSO**

None






# Allocating Structures

Many of the structures passed as parameters to various endpoint functions contain one or more TNetbuf structures. Before a client can call these endpoint functions, the buf and maxlen fields of the TNetbuf must be initialized with a pointer to a buffer area and its length. The length of the buffer will differ for different endpoints. Thus, the client must allocate a buffer big enough to hold the desired data, but not so large that memory is needlessly wasted.

By using the GetEndpointInfo function, a client can determine the required length of buffers. The client can then allocate the buffers and initialize the data structures with pointers to the buffers and their sizes.

Two helper functions make this process easier—Alloc , which allocates a specified structure, and Free, which deallocates, or “frees,” it. The kind of structure to be allocated or freed is passed as a parameter to the functions.

**IMPORTANT:** Alloc and Free are provided for compatibility with XTI. In general, clients should not allocate and free structures on every call, because doing so will degrade client performance. Instead, if structures are to be passed as parameters to endpoint functions, clients can declare the structures just as they would any other variables or data structures. This is especially important for data transfer functions.

# Alloc

**FUNCTION**

`Alloc` Allocate a desired XTI data structure.

**C INTERFACE**

```
void*   OTAlloc(EndpointRef ref, OTStructType structType, UInt32
        fields, OSStatus* err);
```

**C++ INTERFACE**

```
void*   TEndpoint::Alloc(OTStructType structType, UInt32 fields,
        OSStatus* err = NULL);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `structType` | x | / |
| `fields` | x | / |
| `err` | ? | (?) |

`Alloc` allocates one of several data structures for use in subsequent calls. If successful, a pointer to the desired structure is returned. The type of structure that is allocated depends upon the `structType` parameter. The fields parameter indicates which substructures inside the requested structure should also be allocated.

For example, a client that wants to get the protocol address of the endpoint must use the `GetProtAddress` function. In order to make the call, the client must pass in a `TBind` structure whose `addr.buf` field points to a buffer large enough to hold the endpoint's protocol address. The client must use `GetEndpointInfo` to find out how large an address field is required, allocate the memory, and then initialize the `addr.buf` and `addr.maxlen` fields in the `TBind` structure before making the call.

The `Alloc` function handles this work automatically; the client can just make the call:

```
TBind* boundAddr = Alloc(T_BIND, T_ADDR);
```

Table 1-3 shows the structures that can be allocated and the corresponding value to use in the `structType` parameter.






**Table 1-3.** Valid Values for the `structType` parameter of `Alloc` and `Free`

| | |
|---|---|
| `struct TBind` | T_BIND |
| `struct TOptMgmt` | T_OPTMGMT |
| `struct TCall` | T_CALL |
| `struct TDiscon` | T_DIS |
| `struct TUnitData` | T_UNITDATA |
| `struct TUDErr` | T_UDERROR |
| `struct TEndpointInfo` | T_INFO |
| `struct TReply` | T_REPLYDATA |
| `struct TRequest` | T_REQUESTDATA |
| `struct TUnitRequest` | T_UNITREQUEST |
| `struct TUnitReply` | T_UNITREPLY |

Each structure, except `TEndpointInfo`, contains at least one field of type struct `TNetbuf`. For each field of this type, the client may specify that the buffer for that field should be allocated also. The length of the buffer allocated will be at least as large as the appropriate size returned in the `GetEndpointInfo` function. The `fields` parameter is a bitwise-OR of the following constants and specifies which buffers to allocate:

| | |
|---|---|
| T_ADDR | The `addr` field of the `TBind`, `TCall`, `TUDErr`, `TUnitRequest`, `TUnitData`, or `TUnitData` structures. |
| T_OPT | The `opt` field of the `TOptMgmt`, `TCall`, `TUDErr`, `TRequest`, `TReply`, `TUnitRequest`, `TUnitReply`, or `TUnitData` structures. |
| T_UDATA | The `udata` field of the `TCall`, `TDiscon`, `TUnitData`, `TRequest`, `TReply`, `TUnitRequest`, or `TUnitReply` structures. The value of the `udata.maxlen` field depends upon the kind of structure being allocated. |
| T_ALL | All relevant fields of the desired structure will be allocated. |

For each `TNetbuf` allocated, the `maxlen` field of the `TNetbuf` will be set to the length of the buffer allocated, and the `len` field of the `TNetbuf` will be set to zero. Irrelevant or unknown values passed in the `fields` parameter are ignored.

The length of any allocated fields will be appropriate only for this endpoint. Clients should not use the structure pointer that is returned by this function in calls to any other endpoint.

Any `TNetbuf` structures in the requested structure that are not allocated will have their `maxlen`, `len`, and `buf` fields all set to zero.

Any field which has an infinite size defined by the `TEndpointInfo` will not be allocated.

**VALID STATES**

All

**RESULT CODES**

`kOTStructureTypeErr`

`kOTNotSupportedErr`

**SEE ALSO**

`Free, GetEndpointInfo`






# Free

**FUNCTION**

`Free` `Free` a structure previously allocated with the `Alloc` function.

**C INTERFACE**

```
OSStatus    OTFree(void* ptr, OTStructType structType);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Free(void* ptr, OTStructType structType);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ptr` | x | / |
| `structType` | x | / |

`Free` frees the memory a client has allocated using the `Alloc` function. The client must pass in the structure type in the `structType` parameter and a pointer to the structure in the `ptr` field. (For the valid values of the `structType` parameter, see the description of the `Alloc` function.)

The client is responsible for passing the `structType` parameter that exactly matches the type of structure being freed. The client may not change the `buf` field in any of the `TNetbuf` fields of the allocated structure before calling the `Free` function.

**VALID STATES**

All

**RESULT CODES**

`kOTStructureTypeErr`

`kOTNotSupportedErr`

**SEE ALSO**

None

# Managing Options

> Applications can be written that never deal with any options. Skipping this whole section the first time through is recommended. The most important thing to remember about options is that the use of options is completely optional. All endpoints have default values for options.

This section describes the framework for the use of options. This framework is obligatory for all endpoints. The specific options that are legal for use with a specific endpoint are described in the documentation for each specific endpoint. General options are specified in the function description for the `OptionManagement` call.

## Overview

Not all transports are interchangeable, and clients that want to make use of particular feature sets of an endpoint need to have a way to access these features.

The X/Open XTI interface addresses this requirement with a process called `option management`. Open Transport supports the same mechanism through the `OptionManagement` function that allows the client of an endpoint to negotiate options, check for existence of certain options, retrieve the default options, and retrieve the current options.

The format of options is specified, but the value is not. XTI has defined the value of options for many protocols in the ISO and TCP/IP protocol families. Apple and AT&T have defined options for some protocols in the AppleTalk family. The formats of these options can be found in the individual documents for the protocol families.

Free-form options and options specific to the particular kind of endpoint seem to defy one of the goals of Open Transport: that a client can interchangeably use endpoints that provide a similar type of service. There are two features of Open Transport that reduce the size of this problem: default options and configuration. All endpoints have default options that are “good enough” for most uses, so a client need not specify options when making endpoint function calls that take an option parameter.

There will be situations where use of the `OptionManagement` call is unavoidable, but in many cases correct design of endpoint layers can hide these calls from the application that is the ultimate user of lower endpoint.

There are two general categories of options: those that are association-related and those that are not. Association-related options are intimately related to the particular transport connection or datagram transmission. If a calling client specifies such an option, some ancillary information is passed along to the destination endpoint in most cases. The interpretation and further processing of this information are protocol-dependent. For example, in an ISO connection-oriented communication, the calling client may specify quality-of-service parameters on connection establishment. These are processed and possibly lowered by the called endpoint, then passed along to the called (remote) client, who may degrade them again, and finally returned to the calling client.

Options that are not association-related do not contain information destined for the remote transport user. Some have purely local relevance: an option that enables debugging for example. Others influence the transmission. For example, the option for an IP endpoint that sets the time-to-live field. Local options are negotiated solely between the client and endpoint.






The distinction between these two categories of options is visible in the Open Transport interfaces through the following relationship: On output, the functions `Listen`, `RcvUData`, and `RcvURequest` return association-related options only. The functions `RcvConnect` and `RcvUDErr` may return options of both categories. On input, options of both categories may be specified to the `Accept`, `SndUData`, and `SndURequest` functions. The functions `Connect` and `OptionManagement` can process and return both categories of options.

## Portability

An applications programmer who writes Open Transport programs faces two portability aspects:

- Portability across different protocol families
- Portability across different system platforms

Options are intrinsically coupled with a particular protocol or protocol family. Making explicit use of them degrades portability across protocol families.

Different system platforms may offer different option support for the same protocols due to different implementations. The lists of common options described in the `OptionManagement` function and the protocol-specific options described elsewhere are maximal sets but do not necessarily reflect common implementation practice. Different system platforms will implement subsets that suit their needs. Making careless use of options endangers portability across different system platforms.

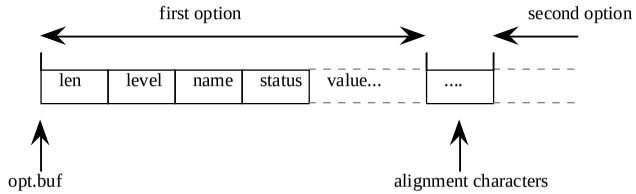
Every implementation of a protocol endpoint can be used with the default values of options. **This means applications can be written that do not care about options at all.**

An application program that processes options retrieved from an Open Transport function should discard options it does not recognize in order to lessen its dependence on different system platforms and future expansion of protocol options and vice versa.

## Option Format

Options are passed to or from an endpoint via an `opt` parameter of type struct `TNetbuf`. Each option in the buffer pointed to by `opt.buf` is of the form `struct TOption`, possibly followed by an option value.

Several options can be concatenated, but each option must start on a long-word boundary:




The `level` field of an option identifies the XTI level (see `GetXTILevel`) of the endpoint, the `name` field identifies the option, and the `len` field contains the total length of the option (including the four long-words and the value fields.). The `status` field is used to indicate the success or failure of an option negotiation (See the `OptionManagement` function for a description). All four of these fields are unsigned longs.

## Option Negotiation

The discussion below describes the general rules governing the passing and retrieving of options and the error conditions that can occur. Unless explicitly restricted, these rules apply to all functions that allow the exchange of options.

## Multiple Options and Options Levels

When multiple options are specified in an option buffer on input, different rules apply to the levels that may be specified, depending upon the function call. Multiple options specified on input to the `OptionManagement` function must address the same option level. Options specified on input to `Connect, Accept, SndUData,` and `SndURequest` can address different levels.

## Illegal Options

Only legal options may be negotiated; illegal options cause failure. An option is illegal if the following applies:

- The length specified in the TOption.`len` exceeds the remaining size of the option buffer (counted from the beginning of the option).
- The option value is illegal. The legal values are defined for each option. (See the documentation specific to the particular endpoint.)

If an illegal option is passed to an endpoint, the following will happen:

- A call to `OptionManagement` will fail with `kOTBadOptionErr`.
- `Accept` or `Connect` will fail with either `kOTBadOptionErr`, or the connection establishment aborts, depending upon the implementation and the time the illegal option is detected. If the connection aborts, a T_DISCONNECT event occurs, and a synchronous call to `Connect` fails with `kOTLookErr`. It depends upon timing and implementation conditions whether an `Accept` can still succeed or whether it fails with `kOTLookErr`.
- A call to `SndUData` either fails with `kOTBadOptionErr` or it successfully returns, but a T_UDERR event occurs to indicate that the datagram was sent.

If the client passes multiple options in one call and one of them is illegal, the call fails as described above. However, it is possible that some or all of the submitted legal options were successfully negotiated. The client can check the current status by calling the `OptionManagement` function with the T_CURRENT action flag set.

Specifying an option level unknown to an endpoint does not cause failure in calls to `Connect`, `Accept`, `SndUData`, or `SndURequest`; the option is ignored. The function `OptionManagement` will fail with `kOTBadOptionErr` if passed an unknown option level.

Specifying an option name that is unknown to or not supported by the endpoint selected by the option level does not cause failure. The option is discarded in calls to `Connect`, `Accept`, `SndUData`, or `SndURequest`. The function `OptionManagement` returns T_NOTSUPPORT in the `level` field of the option.

## Initiating an Option Negotiation

A client initiates an option negotiation when calling `Connect`, `SndUData`, `SndURequest`, or `OptionManagement` with the action flag T_NEGOTIATE set.






The negotiation rules for these functions depend on whether an option request is an absolute requirement or not. This is explicitly defined for each option (see documentation for each specific endpoint kind). For example, in the case of an ISO endpoint, the option that requests use of expedited data is not an absolute requirement, but the option that requests protection could be an absolute requirement.

If the proposed option is an absolute requirement, three outcomes are possible:

- The negotiated option value is the same as the proposed one. When the result of the negotiation is retrieved, the `status` field in the TOption structure is set to T_SUCCESS.
- The negotiation is rejected if the option is supported, but the proposed value cannot be negotiated. This leads the following:

  – `OptionManagement` successfully returns, but the returned option has its `status` field set to T_FAILURE.

  – Any attempt to establish a connection aborts; a T_DISCONNECT event occurs, and a synchronous call to `Connect` fails with `kOTLookErr`.

  – `SndUData` fails with `kOTLookErr` or successfully returns, but a T_UDERR event occurs to indicate that the datagram was not sent.

  If multiple options are submitted in one call, and one of them is rejected, the endpoint behaves as described above. Although the connection establishment, the datagram transmission, or the transaction request fails, options successfully negotiated before some option was rejected retain their negotiated values. There is no undo mechanism.

  The function `OptionManagement` attempts to negotiate each option. The `status` fields of the returned options indicate success (T_SUCCESS) or failure (T_FAILURE).
- If the endpoint does not support the option at all, `OptionManagement` reports T_NOTSUPPORT in the `status` field. The `Connect`, `SndUData`, and `SndURequest` functions ignore the option.

If the proposed option value is not an absolute requirement, two outcomes are possible:

- The negotiated value is of equal or lesser quality than the proposed one (that is, a delay may become longer).

  When the result of the negotiation is retrieved, the `status` field in TOption is set to T_SUCCESS if the negotiated value equals the proposed one, or T_PARTSUCCESS otherwise.
- If the endpoint does not support the option at all, `OptionManagement` reports T_NOTSUPPORT in the `status` field. The functions `Connect`, `SndUData`, and `SndURequest` ignore the option.

Unsupported options do not cause functions to fail or a connection to abort, since different implementations possibly implement different subsets of options. Future enhancements might add additional options that are unknown to earlier implementations of an endpoint. The decision whether or not the missing support of an option is acceptable is left to the client.

The endpoint does not check for multiple occurrences of the same option, possibly with different values. It simply processes the options one after another. However, the client should not make any assumption about the order of processing in case this changes in the future.

Not all options are independent of one another. A requested option value might conflict with the value of another option that was specified in the same call or is currently effective. These conflicts may not be detected at once, but later they might lead to unpredictable results. If detected at negotiation time, these conflicts are resolved within the rules stated above. The outcomes may be quite different and depend upon whether absolute or non absolute requests are involved in the conflict.

Conflicts are usually detected at the time a connection is established or a datagram is sent. If the options are negotiated with `OptionManagement`, conflicts are usually not detected at this time, since independent processing of the requested options must allow for temporal inconsistencies.

When called, the functions `Connect, SndUData,` and `SndURequest` initiate a negotiation of all association-related options according to the rules of this section. Options not explicitly specified in the function calls are taken from a (logical) internal option buffer containing default values, configured values, or values of a previous negotiation.

## Responding to a Negotiation Proposal

In connection-oriented communication, some protocols give the peer transport client the opportunity to negotiate characteristics of the connection to be established. These characteristics are association-related options. With the connect indication, the called client receives (via a `Listen` function) a proposal about the option values that should be effective for this connection. The called user can accept this proposal or weaken it by choosing values of lower quality. The called client can, of course, refuse the connection establishment altogether.

For connection-oriented endpoints, the called user responds to a negotiation proposal via `Accept`. If the called endpoint client tries to negotiate an option of higher quality than proposed, the outcome depends on the protocol to which that option applies. Some protocols may reject the option, some protocols take other appropriate action described in protocol-specific documentation. If an option is rejected, the following error occurs:

> The connection fails; a T_DISCONNECT event occurs. It depends on timing and implementation conditions whether the `Accept` call still succeeds or fails with a `kOTLookErr` result.

If multiple options are submitted with `Accept` and one of them is rejected, the connection fails as described above. Options that could be successfully negotiated before the erroneous option was detected retain their negotiated value. There is no undo mechanism.

The response options can either be specified with the `Accept` call, or can be preset by the responding endpoint (not the listening endpoint!) in an `OptionManagement` call (with the T_NEGOTIATE action flag) prior to `Accept`. Note that the response to a negotiation proposal is activated when `Accept` is called. An `OptionManagement` call with erroneous option values as described above will succeed; the connection aborts at the time the `Accept` is called.

The connection also fails if the selected option values lead to contradictions.

The function `Accept` does not check for multiple specifications of the same option. Unsupported options are ignored.

## Retrieving Information About Options

This section describes how a client can retrieve information about options. To be explicit, a client must be able to

- know the result of a negotiation (i.e. at the end of a connection establishment)






- know the proposed option values under negotiation (during connection establishment)
- retrieve option values sent by the remote client for notification only
- check option values currently effective for the transport endpoint

So that the client can perform these operations, the `Connect, Listen, OptionManagement, RcvConnect, RcvUData, RcvUDErr,` and `RcvURequest` functions take an output parameter `opt` of type struct `TNetbuf`. The client must specify a buffer address where the options will be written to in `opt.buf`, and `opt.maxlen` must contain the buffer's size. The client may set `opt.maxlen` to zero to indicate that no options are to be retrieved.

Which options are returned depends upon the function call:

`Connect` (synchronous) and `RcvConnect`

> The function returns the values of all association-related options that were received with the connection response and the negotiated values of those non-association-related options that had been specified on input. However, options specified on input to the `Connect` call that are not supported, or refer to an unknown option level are discarded and not returned on output.
>
> The `status` field of each option returned with `Connect` or `RcvConnect` indicates if the proposed value (T_SUCCESS) or a degraded value (T_PARTSUCCESS) has been negotiated. The `status` field of received ancillary information that is not subject to negotiation is always set to T_SUCCESS.

`Listen`

> The received association-related options are related to the incoming connection (identified by the sequence number), not to the listening endpoint. (However, the option values currently effective for the listening endpoint can affect the values retrieved by `Listen`, since the endpoint might be involved in the negotiation process, too). Thus, if the same options are specified in a call to `OptionManagement` with the action flag set to T_CURRENT, `OptionManagement` will usually not return the same values.
>
> The number of received options may be variable for subsequent connection indications, since many association-related options are transmitted only on explicit demand by the calling client. It is even possible that no options at all are returned.

`RcvUData`

> The received association-related options are related to the incoming datagram, not to the endpoint. Thus, if the same options are specified in a call to `OptionManagement` with the action field set to T_CURRENT, `OptionManagement` will usually not return the same values.
>
> The number of options received may vary from call to call.
>
> The `status` field is irrelevant.

`RcvUDErr`

> The returned options are related to the options input at the previous `SndUData` call that produced the error. Which options are returned and which values they have depend on the specific error condition.

The `status` field is irrelevant.

`OptionManagement`

This call can process and return both categories of options. It acts on options related to the endpoint, not on options related to a connect indication or an incoming datagram.

## Privileged and Read-Only Options

Privileged options or option values are those that may be requested by privileged clients only. The meaning of privileged is implementation-defined.

Read-only options serve for information purposes only. The client may be allowed to read the option value but not to change it. For example, to select the value of a protocol timer or the maximum length of a protocol data unit may be too subtle to leave to the client, though the knowledge about this value may be of some interest. An option might be read-only for all clients or solely for non-privileged clients. A privileged option might be inaccessible or read-only for non-privileged users.

An option might be negotiable in some XTI states and read-only in other XTI states. For example, the ISO quality-of-service options are negotiable in the T_IDLE and T_INCON states, and read-only in all other states (except T_UNINIT).

If a client requests negotiation of a read-only option, or a non-privileged client requests illegal access to a privileged option, the following outcomes are possible:

- `OptionManagement` successfully returns, but the returned option has its `status` field set to T_NOTSUPPORT if a privileged option was requested illegally, and T_READONLY if the modification of a read-only option was requested.
- If negotiation of a read-only option is requested, `Accept` or `Connect` fail with `kOTAccessErr`, or the connection establishment aborts, and a T_DISCONNECT event occurs. If the connection aborts, a synchronous call to `Connect` will fail with `kOTLookErr`. If a privileged option is illegally requested, the option is quietly ignored.(A non-privileged client shall not be able to select an option which is privileged or unsupported.) It depends on timing and implementation conditions whether an `Accept` call still succeeds or fails with `kOTLookErr`.
- If negotiation of a read-only option is requested, `SndUData` may return `kOTLookErr` or successfully return, but a T_UDERR event occurs to indicate that the datagram was not sent. If a privileged option is illegally requested, the option is quietly ignored. (A non-privileged client will not be able to select an option that is privileged or unsupported.)

If multiple options are submitted to `Connect, Accept, SndUData`, or `SndURequest`, and a read-only option is rejected, the connection or the transmission fails as described. Options that could be successfully negotiated before the erroneous option was discovered retain their negotiated values. There is no undo mechanism.






## Option Management of a Transport Endpoint

This section describes how option management works during the lifetime of a transport endpoint.

Each transport endpoint is (logically) associated with an internal option buffer. When a transport endpoint is created, this buffer is filled with a system default value for each supported option. Depending upon the option, the default may be “Option Enabled” or “Option Disabled” or denote a time span, etc. These default settings are appropriate for most uses. Whenever an option value is modified in the course of an option negotiation, the modified value is written to this buffer and overwrites the previous one. At any time, the buffer contains all option values that are currently effective for this endpoint.

The current value of an option can be retrieved at any time by calling `OptionManagement` with the T_CURRENT action flag set. Calling `OptionManagement` with the T_DEFAULT action flag set yields the system default for the specified option.

A transport client can negotiate new option values by calling `OptionManagement` with the T_NEGOTIATE action flag set. The negotiation follows the rules in a previous section, “Option Negotiation”.

Some options may be modified only in specific XTI states and are read-only in other XTI states. Many association-related options, for example, may not be changed in the state T_DATAXFER, and an attempt to do so will fail. The legal states for each option are specified in the documentation for the option.

Association-related options take effect at the time a connection is established or a datagram is transmitted. This is the case if they contain information that is transmitted across the network or they determine specific transmission characteristics. If such an option is modified by a call to `OptionManagement`, the endpoint checks whether the option is supported and negotiates a value according to its current knowledge. This value is written to the internal option buffer. The final negotiation takes place if the connection is established or the datagram is transmitted. This can result in a degradation of the option value or even in a negotiation failure. The negotiated values are written to the internal option buffer.

Some options may be changed in the state T_DATAXFER, for example those specifying buffer sizes. Such changes might affect the transmission characteristics and lead to unexpected side effects, such as data loss if a buffer size was shortened.

The endpoint client can explicitly specify both categories of options on input when calling `Connect, Accept, SndUData, or SndURequest`. The options are first locally negotiated option-by-option, and the resulting values written to the internal option buffer. The modified option buffer is then used if a further negotiation step across a connection is required. The newly negotiated values are then written to the internal option buffer.

At any stage, a negotiation failure can lead to an abort of the transmission. If a transmission aborts, the option buffer will preserve the content it had at the time the failure occurred. Options that could be negotiated just before the error occurred are retained in the option buffer whether or not the function call fails or succeeds.

It is up to the endpoint user to decide which options to specify on input when calling `Connect, Accept, SndUData, or SndURequest`. The client need not pass options at all by setting the `len` field of the functions input `opt` parameter to zero. The current content of the internal option buffer is then used for negotiation.

The negotiation procedure for options at the time a `Connect, Accept, SndUData, or SndURequest` function is made is the same as described in earlier sections whether the options are explicitly specified or are implicitly taken from the internal option buffer.

The transport client must not make assumptions about the order in which options are processed during negotiation.

A value in the option buffer is modified only as a result of successful negotiation of this option. It is not changed by a connection release. There is no history mechanism that would restore the buffer state existing prior to the connection establishment or the datagram transmission. The transport client must be aware that a connection establishment or datagram transmission may change the internal option buffer, even if each option was originally initialized to its default value.

## The Option value `T_UNSPEC`

Some options may not have a fully specified value all the time. When an option does not have a value, it gets the value T_UNSPEC in the `status` field.

An endpoint may also return the value T_UNSPEC if it cannot currently access the option value. This may happen, for example, in the state T_UNBND in systems where the protocol stack resides on a separate host. An endpoint will never return T_UNSPEC if the option is not supported at all.

If T_UNSPEC is a legal value for a specific option, it may be used by the client on input, too. It is used to indicate that it is left up to the endpoint to choose an appropriate value. This is especially useful in complex options such as ISO throughput where the option value has an internal structure. The endpoint client may leave some fields unspecified by selecting this value. If the client proposes T_UNSPEC, the endpoint is free to select an appropriate value. This might be the default value, some other explicit value, or T_UNSPEC.

The documentation for each option will specify whether or not T_UNSPEC is a legal value for negotiation purposes.

## The `info` Argument

The functions `OpenEndpoint, AsyncOpenEndpoint`, and `GetEndpointInfo` return values representing characteristics of the transport endpoint in their info parameter. The value of `info->options` is used in he `Alloc` function to allocate storage for an option buffer to be used in a endpoint function call. The value is sufficient for all uses.

In general, `info->options` also includes the size of privileged options, even if these are not read-only for non-privileged users. Alternatively, an implementation can choose to return different values in `info->options` for privileged and non-privileged users.

The values in `info->etsdu, info->tsdu, info->connect`, and `info->discon` possibly diminish as soon as the T_DATAXFER state is entered. Calling `OptionManagement` does not influence these values.

## Summary

- The value of an option is defined by a header struct TOption, followed by an option value.
- On input, several options can be specified in an input `opt` parameter. Each option must begin on a long-word boundary.
- An endpoint is (logically) associated with an internal option buffer, where the currently effective value are stored. Each successful negotiation of an option modifies this buffer, regardless of whether the call initiating the negotiations succeeds or fails.






- When calling `Connect, Accept, SndUData,` and `SndURequest`, the client can choose to use the current options by setting the `len` field of the input `opt` parameter to zero.
- If a connection is accepted with the `Accept` function, the explicitly specified option values together with the currently effective options of the endpoint accepting the connection matter. But, only in the case where the `Accept` function is instructed to accept the connection on an endpoint different from the one that received the connection request.
- The options returned by `RcvUDErr` are those negotiated with the outgoing datagram that produced the error. If the error occurred during the option negotiation, the returned option might represent some mixture of partly negotiated and not-yet-negotiated options.

# OptionManagement

**FUNCTION**

`OptionManagement` Manage options for an endpoint.

**C INTERFACE**

```
OSStatus    OTOptionManagement(EndpointRef ref, TOptMgmt* req, TOptMgmt*
            ret);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::OptionManagement(TOptMgmt* req, TOptMgmt* ret);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->opt.maxlen` | / | / |
| `req->opt.len` | `x >= 0` | / |
| `req->opt.buf` | (x) | / |
| `req->flags` | `x >= 0` | / |
| `ret->opt.maxlen` | x | / |
| `ret->opt.len` | / | x |
| `ret->opt.buf` | ? | (?) |
| `ret->flags` | / | `x >= 0` |

`OptionManagement` allows a client to retrieve, verify, or negotiate protocol options with the endpoint. The `req` and `ret` parameters point to `TOptMgmt` structures containing these members:

```
struct TNetbuf opt;
OTFlags flags;
```

The `opt` field identifies the protocol options, and the `flags` field is used to specify the action to take with the options.

The `flags` field of the `req` parameter is used to request a specific action of the endpoint, and the `opt` field is used to pass the value of the options to the endpoint.

Each option in the options buffer is of the form struct TOption possibly followed by an option value. (See the earlier section “Option Formats”.)

The `level` field in each struct TOption identifies the XTI level of the transport provider. The `name` field identifies the option within the level, and `len` contains the total length of the option (that is, the length of the TOption structure plus the length of the option value). If `OptionManagement` is called with the action T_NEGOTIATE set, the status field of the returned options contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start on a long-word boundary. A macro defined in the `OpenTransport.h` header file, `OPT_NEXTHDR(pbuf, buflen, poption)` can be used by the client for this purpose. The parameter `pbuf` denotes a pointer to an option buffer, `opt.buf`, and `buflen` is its length. The parameter `poption` points to the current option in the option buffer. `OPT_NEXTHDR` returns a pointer to the position of the next option or returns a null pointer if the option buffer is exhausted. The macro is helpful for both reading and writing.

If the client specifies several options on input, all options must address the same `level`.






If any option in the options buffer does not indication the same `level` as the first option, or the level specified is unsupported, then the `OptionManagement` request will fail with `kOTBadOptionErr`. If the error is detected, some options have possibly been successfully negotiated. The client can check the current status by calling `OptionManagement` with the T_CURRENT option flag set.

The `flags` field of the `TOptMgmt` structure pointed to by the `req` parameter must specify one of the following actions:

T_NEGOTIATE

This action enables the client to negotiate the values of the options with the endpoint. The endpoint will evaluate the requested options, negotiate the values, and return the resulting values in the `TOptMgmt` structure pointed to by the `ret` parameter. The `status` field of each returned option is set to indicate the result of the negotiation. The value is T_SUCCESS if the proposed value was negotiated, T_PARTSUCCESS if a degraded value was negotiated, T_FAILURE if the negotiation failed (according to the negotiation rules), T_NOTSUPPORT if the transport provider does not support this option or illegally requests negotiation of a privileged option, and T_READONLY if modification of a read-only option was requested. If the `status` is T_SUCCESS, T_FAILURE, T_NOTSUPPORT, or T_READONLY, the returned option value is the same as the one requested on input.

On return, the overall result of the negotiation is returned in `ret->flags`. This field contains the worst single result; the rating from worst to best is: T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS.

For each level, the option T_ALLOPT (see below) can be requested on input. No value is given with this option; only the TOption part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in `ret->opt.buf`. (Note that depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.)

T_CHECK

This action enables the client to verify whether the options specified in the `req` `TOptMgmt` structure are supported by the endpoint.

If an option is specified with no option value (it consists only of a TOption structure), the option is returned with its status field set to T_SUCCESS if it is supported, T_NOTSUPPORT if it is not supported or needs additional client privileges, and T_READONLY if it is read-only (in the current XTI state). No option value is returned.

If an option is specified with an option value, the status field of the returned option has the same value as if the user had tried to negotiate this value with T_NEGOTIATE. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT, or T_READONLY, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in `ret->flags`. This field contains the single worst result of the option checks; the rating is the same as for the T_NEGOTIATE action flag.

Note that no negotiation takes place. All currently effective options remain unchanged.

T_DEFAULT — This action enables a client to retrieve the default options supported by the endpoint. The client specifies the options of interest in `req->opt.buf`. The option values are irrelevant and will be ignored; it is sufficient to specify the TOption part of an option only. The default values are then returned in `ret->opt.buf`.

The status field returned is T_NOTSUPPORT if the protocol level does not support this option, or the client illegally requested a privileged option, T_READONLY if the option is read-only, and T_SUCCESS in all other cases. The overall result of the request is returned in ret->flags. This field contains the single worst result; the rating is the same as for the T_NEGOTIATE flag.

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their default values are returned. In this case, `ret->opt.maxlen`, must be given at least the value in `info->options.` (See `GetEndpointInfo`) before the call.

T_CURRENT — This action enables a client to retrieve the currently active options. The client specifies the options of interest in `req->opt.buf`. The option values are irrelevant and will be ignored; it is sufficient to specify the TOption part of an option only. The current values are then returned in `ret->opt.buf`.

The status field returned is T_NOTSUPPORT if the protocol level does not support this option, or the client illegally requested a privileged option, T_READONLY if the option is read-only, and T_SUCCESS in all other cases. The overall result of the request is returned in ret->flags. This field contains the single worst result; the rating is the same as for the T_NEGOTIATE flag.

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their current values are returned. In this case, `ret->opt.maxlen`, must be given at least the value in `info->options.` (See `GetEndpointInfo`) before the call.

The option T_ALLOPT can be used only with the `OptionManagement` call, and even then, only with the actions T_NEGOTIATE, T_DEFAULT, and T_CURRENT. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a TOption only. Since in a `OptionManagement` call, only options of one level may be addressed, this option should not be requested together with other options. The function returns as soon as this option is processed.






Other options are processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Endpoints may not be able to provide an interface capable of supporting the T_NEGOTIATE and/or the T_CHECK actions. In this case the `kOTNotSupportedErr` error is returned.

For an endpoint in synchronous mode, the endpoint will return results of the option management in the `TOptMgmt` structure pointed to by the `req` parameter.

If the endpoint is in asynchronous mode, a notification routine has been installed, and the `OptionManagement function` returns `kOTNoError`, a `T_OPTIONMGMTCOMPLETE` event will be issued when the function completes. The result parameter will be `kOTNoError` if the function completed successfully. Otherwise, it will contain a result code describing the reason that the function failed. The `cookie` parameter passed to the notification routine to indicate completion contains the value of the `ret` parameter that was passed to the original function call.

If a notification routine has not been installed, it is not possible to determine when this command is completed.

While an `OptionManagement` call is outstanding, any other functions that are called for the same endpoint will return with a `kOTStateChangeErr` result code.

**Note:** In asynchronous mode, the T_OPTIONMGMTCOMPLETE event may be issued before the `OptionManagement` function returns to the client.

**XTI-LEVEL OPTIONS**

XTI-level options are not specific to a particular endpoint. An XTI implementation supports none, all, or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.

The options below are not association-related. They may be negotiated in all XTI states. The protocol level for all of these options is XTI_GENERIC.

| option name | type of option value | legal option value | meaning |
|---|---|---|---|
| XTI_DEBUG | array of unsigned longs | see text | enable debugging |
| XTI_LINGER | struct linger | see text | linger on close if data present |
| XTI_RCVBUF | unsigned long | size in bytes | receive buffer size |
| XTI_RCVLOWAT | unsigned long | size in bytes | rcv low-water mark |
| XTI_SNDBUF | unsigned long | size in bytes | send buffer size |
| XTI_SNDLOWAT | unsigned long | size in bytes | send low-water mark |

A request for XTI_DEBUG is an absolute requirement. A request to activate XTI_LINGER is an absolute requirement; the timeout value to this option is not. XTI_RCVBUF, XTI_RCVLOWAT, XTI_SNDBUF, XTI_SNDLOWAT are not absolute requirements.

XTI_DEBUG — This option enables debugging. The values of this option are implementation defined. Debugging is disabled if the option is specified with no value.

XTI_LINGER — This option is used to linger the execution of a `CloseProvider` if data is still queued in the send buffer. The option value specifies the linger period. If `CloseProvider` is issued, and the send buffer is not empty, the endpoint attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

CloseProvider will immediately return and the endpoint holds the connection open for at most the linger period.

The option value consists of a t_linger structure:

```
struct t_linger
{
        long l_onoff;  // switch option on/off
        long l_linger;        // linger period in seconds
};
```

The legal values for the l_onoff field are:

T_NO switch the option off

T_YES activate option

The value of l_onoff is an absolute requirement.

The l_linger field determines the linger period in seconds. The client can request the default value by setting the field to T_UNSPEC. The default timeout values depend upon the endpoint (It is often T_INFINITE.) Legal values for this field are T_UNSPEC, T_INFINITE, and all non-negative numbers.

The l_linger field is not an absolute requirement. An implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note that this option does not linger the execution of SndDisconnect.

XTI_RCVBUF This option is used to adjust the internal buffer size allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. An implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_RCVLOWAT This option is used to set a low-water mark in the receive buffer. The option value gives the minimal number of bytes that must have accumulated in the receive buffer before they become visible to the client. If and when the amount of accumulated received data exceeds the low-water mark, a T_DATA event is issued. The client may then read the data with Rcv or RcvUData.

This request is not an absolute requirement. An implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SDNBUF This option is used to adjust the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. An implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDLOWAT This option is used to set a low-water mark in the send buffer. The option value gives the minimal number of bytes that must have accumulated in the send buffer before they are sent.






This request is not an absolute requirement. An implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

**VALID STATES**

All

**RESULT CODES**

kOTAccessErr

kOTBadFlagErr

kOTBadOptionErr

kOTCanceledErr

kOTNotSupportedErr

**SEE ALSO**

OTCreateOptions, OTCreateOptionString, Accept, Alloc, Connect, GetEndpointInfo, Listen, OpenEndpoint, RcvConnect

# Using Connectionless Datagrams

The functions in this section apply only to connectionless datagram endpoints. A connectionless datagram endpoint usually provides a datagram service. Protocols such as DDP, IP, PPP, 802.2 Type 1, 802.3, etc., are connectionless endpoints.

The `SndUData` function is used to send data on a connectionless datagram. Each `SndUData` function call requires the protocol address of the destination of the datagram.

Some endpoint implementations do not detect an error in the attempt to send a datagram until after the `SndUData` function has already returned successfully. In this case, the endpoint will issue a T_UDERR event and the client can determine the particular error by issuing the `RcvUDErr` function.

The `RcvUData` function is used to read incoming datagrams. Each datagram read will have a remote protocol address (the source) associated with it.

Both the `RcvUData` and `SndUData` functions are supported only when the endpoint is bound and in the T_IDLE state.






# SndUData

**FUNCTION**

`SndUData` Send a data unit.

**C INTERFACE**

```
OSStatus    OTSndUData(EndpointRef ref, TUnitData* udata);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndUData(TUnitData* udata);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `udata->addr.maxlen` | / | / |
| `udata->addr.len` | `x >= 0` | / |
| `udata->addr.buf` | (x) | / |
| `udata->opt.maxlen` | / | / |
| `udata->opt.len` | `x >= 0` | / |
| `udata->opt.buf` | (?) | / |
| `udata->udata.maxlen` | / | / |
| `udata->udata.len` | x | / |
| `udata->udata.buf` | (x) | / |

`SndUData` is used on connectionless datagram endpoints to send a data unit. The `udata` parameter points to a `TUnitData` structure containing the following fields:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
```

The `addr` field specifies the protocol address of the destination, `opt` identifies any endpoint-specific options that the client wants to use for this request, and `udata` specifies the client data to be sent. The client may choose not to specify protocol options by setting the `len` field inside the `opt TNetbuf` to zero.

The `udata` member of the `udata` parameter contains the data to be sent. If the `udata->udata.len` field is zero, and sending of zero data bytes is not supported by the endpoint, the `kOTBadDataErr` error will be generated.

If the amount of data in udata exceeds the current TSDU size, a `kOTBadDataErr` error will be generated.

A client may send non-contiguous data by setting the `udata->udata.buf` pointer to point to an `OTData` structure, and setting the udata->udata.len value to `kNetbufDataIsOTData`

It is not possible for all endpoints to detect the conditions that result in the `kOTBadAddressErr` or `kOTBadOptionErr` errors. The existence of these errors is signaled by a `T_UDERR` event, and the client can read them by making the `RcvUDErr` function call.

The client may negotiate the XTI_SNDLOWAT option with an endpoint using the `OptionManagement` function. This option value gives the minimal number of bytes that must have accumulated in the endpoint's send buffer before they are sent. Not all endpoints support the XTI_RCVLOWAT option.

If the endpoint is in non-blocking or asynchronous mode, the `SndUData` function will return a kOTFlowErr if flow control restrictions prevent the data from being accepted by the transport provider at the time the function is issued. After this error occurs, a T_GODATA event will be issued when the flow control restrictions are lifted. This error will never be returned if the endpoint is in blocking mode.

The behavior of SndUData is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | `kOTFlowErr` never returned<br>Returns when flow control lifts |
| **Sync/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately |
| **Async/Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately |
| **Async/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately |

**VALID STATES**

T_IDLE

**RESULT CODES**

`kOTAccessErr`
`kOTBadAddressErr`
`kOTBadOptionErr`
`kOTCanceledErr`
`kOTFlowErr`
`kOTLookErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

`AckSends`, `DontAckSends`, `RcvUDErr`






# RcvUDErr

**FUNCTION**

`RcvUDErr` Read the error result from a previous call to the `SndUData` function.

**C INTERFACE**

```
OSStatus    OTRcvUDErr(EndpointRef ref, TUDErr* uderr);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvUDErr(TUDErr* uderr);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `uderr->addr.maxlen` | x | / |
| `uderr->addr.len` | / | x |
| `uderr->addr.buf` | ? | (?) |
| `uderr->opt.maxlen` | x | / |
| `uderr->opt.len` | / | x |
| `uderr->opt.buf` | ? | (?) |
| `uderr->error` | / | x |

`RcvUDErr` is used with connectionless datagram endpoints to receive an error result on a previously sent data unit, and should be issued only after the endpoint has detected a unit data error and has issued the T_UDERR event.

The client passes a pointer to a `TUDErr` structure containing the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
SInt32 error;
```

Before calling `RcvUDErr`, you must set the `maxlen` fields of `addr` and `opt`, to indicate the maximum size of each buffer. If the size of a result exceeds that of one of these buffers, `RcvUDErr` returns the result code `kOTBufferOverflowErr`. The error indication, however, is nonetheless cleared.

When `RcvUDErr` returns, the `udata->addr` structure specifies the destination protocol address of the erroneous data unit, the `udata->opt` structure identifies protocol-specific options that were associated with the data unit, and `udata->error` specifies a protocol-dependent result code.

If the client does not care to identify the data unit that produced the error, the `uderr` parameter may be set to `NULL`, and `RcvUDErr` will clear the error indication without reporting any information to the client.

**VALID STATES**

T_IDLE

**RESULT CODES**

kOTBufferOverflowErr
kOTNotSupportedErr
kOTNoUDErrErr
kOTOutStateErr

**SEE ALSO**

SndUData, RcvUData






# RcvUData

**FUNCTION**

RcvUData Read a data unit.

**C INTERFACE**

```
OSStatus    OTRcvUData(EndpointRef ref, TUnitData* udata, OTFlags*
            flags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvUData(TUnitData* udata, OTFlags* flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| udata->addr.maxlen | x | / |
| udata->addr.len | / | x |
| udata->addr.buf | ? | (?) |
| udata->opt.maxlen | x | / |
| udata->opt.len | / | x |
| udata->opt.buf | ? | (?) |
| udata->udata.maxlen | x | / |
| udata->udata.len | / | x |
| udata->udata.buf | ? | (?) |
| flags | / | x |

RcvUData is used by a connectionless datagram client to receive a data unit. The client passes a pointer to a TUnitData structure parameter, udata, to hold information associated with the received data unit, and flags is set on return to indicate that the complete data unit was not received. The TUnitData structure has the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
```

The maxlen fields of addr, udata and opt must be set before calling this function. The maxlen field of the addr and opt indicates the maximum size of each buffer.

On return from this call, addr contains the remote protocol address of the data unit, opt contains protocol-specific options that were associated with the data unit, and udata contains the user data that was received.

If the endpoint is in synchronous blocking mode, the endpoint will wait for data if none is currently available. Generally, this method of operation is discouraged as it may lead to a 'hang' if no data ever becomes available. If the client is doing other operations in synchronous mode, it should call SetNonBlocking before calling RcvUData, to prevent the RcvUData call from waiting indefinitely.

If the endpoint is in asynchronous mode or is not blocking, the function will fail with the `kOTNoDataErr` result if no data is available. If a notification routine has not been installed on the endpoint, a client may poll for the arrival of a data unit by calling `Look` and checking for the T_DATA event flag. Or, if the client has installed a notification routine on the endpoint, a T_DATA event will be passed to the notification routine (See `InstallNotifier`). Additionally, once a client gets the T_DATA event, it should not expect to get another T_DATA event until making the `RcvUData` call returns either a `kOTNoDataError` or `kOTLookErr` error.

Clients should be prepared for a T_DATA event and then a `kOTNoDataErr` error when a `RcvUData` call is made. This seems unusual, but may occur as endpoints reclaim unread data in low memory conditions (for unreliable endpoints, only).

If the buffer defined in the `udata` field of the `TUnitData` structure is not large enough to hold the current data unit, the buffer will be filled and T_MORE will be set in `flags` on return to indicate that another `RcvUData` call should be made to retrieve the rest of the data unit. Subsequent calls to `RcvUData` will return zero for the length of the address and options until the full data unit has been received.

**VALID STATES**

T_IDLE

**RESULT CODES**

`kOTBufferOverflowErr`
`kOTCanceledErr`
`kOTLookErr`
`kOTNoDataErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

`SetAsynchronous`, `SetBlocking`, `SetNonBlocking`, `SetSynchronous`, `SndUData`






# Using Connections

The section that follow describe how a client uses the Open Transport calls to establish a connection between two endpoints. This section only applies to connection-oriented endpoints. A client can either actively initiate a connection, or it can passively wait for an endpoint to receive an incoming connection request. Additionally, a client can wait for incoming connection requests on one endpoint and accept the connection on a different endpoint.

## Initiating a connection

Before initiating a connection on an endpoint, the client must first bind the endpoint with the `Bind` function. The client then uses the `Connect` function to initiate the connection. The parameters to the connect call include the address of the remote connection end, any data to send along with the connection request (if the particular protocol the endpoint implements allows it), and any connection options that the client wants to specify.

### Synchronous Mode

If the endpoint is in synchronous mode, the connect function will not return until the connection has either been established, or the connection attempt has failed. If the connection succeeds, the function will return zero.

If the connection does not succeed, the function returns `kOTLookErr`. There will be a T_DISCONNECT event pending, and the event can be cleared by issuing the `RcvDisconnect` function.

The sequence below shows the order of events for a successful connection opening.

| **Local** | **Remote** |
|---|---|
| Client makes a `Connect` call. | |
| | Remote end accepts the connection request. |
| Client's `Connect` call returns with no error. | |

If remote end rejects the connection request, or the connection request fails in some other way, this is the sequence of events, for all synchronous endpoints:

| **Local** | **Remote** |
|---|---|
| Client makes a `Connect` call. | |
| | Remote end rejects or ignores the connection request. |
| Client's `Connect` call returns with `kOTLookErr` | |
| Client issues `RcvDisconnect` call. | |

## Asynchronous Mode

If the endpoint is in asynchronous mode, the connect function will return a result code of `kOTNoDataErr`.

When the connection is successfully established or the connection attempt fails, the Open Transport Library will call the client's notification routine with a T_CONNECT event and pass the same value that the client passed in the `Connect` call's `rcvCall` parameter as the `cookie` parameter .

If the connection attempt fails, then there is also a pending T_DISCONNECT event, and the client must call `RcvDisconnect` to clear this event.

The sequence below shows the order of events for a successful connection opening.

| **Local** | **Remote** |
|---|---|
| Client makes a `Connect` call. It returns with `kOTNoDataErr`. | |
| | Remote end accepts the connection request. |
| Client's notification routine is called with a T_CONNECT event. The result code passed into the notification routine will be `kOTNoError`. | |
| The client calls `RcvConnect`. The endpoint will change state to T_DATAXFER. | |

If remote end rejects the connection request, this is the sequence of events:

| **Local** | **Remote** |
|---|---|
| Client makes a `Connect` call. It returns with `kOTNoError`. | |
| | Remote end rejects the connection request, or the request fails. |
| Client's notification routine is called with a T_DISCONNECT event | |
| The client calls `RcvDisconnect`. | |






If some other error occurs, this will be the sequence of events:

| **Local** | **Remote** |
|---|---|
| Client makes a `Connect` call. It returns with `kOTNoError`. | |
| Client's notification routine is called with a T_CONNECT event with an error in the result code. | |

# Waiting for a Connection

The client prepares an endpoint for handling incoming connection requests by specifying a non-zero, positive value for the `qlen` field of the `TBind` structure passed into the `Bind` function. This causes the endpoint to start listening for incoming connection requests.

When a connection request arrives, the Open Transport Library will issue a T_LISTEN event to the client's notification routine to indicate that an incoming connection request has arrived. The client must issue the `Listen` function to retrieve the information associated with the connection request. This information includes the remote address, any options associated with the request, any data associated with the request, and a sequence number. The client must store this sequence number until the client has accepted or rejected the request.

The client can either reject the incoming connection request by calling the `SndDisconnect` function, or the client can accept the incoming connection by calling the `Accept` function. In both cases, the client must pass the sequence number returned from the `Listen` function to indicate which connection indication should be rejected or accepted.

If the `qlen` field in the `TBind` structure the client passed in the `Bind` function as the `reqAdd` parameter is greater than one, and the `qlen` field in the `TBind` structure passed as the `retAddr` parameter and filled in by the `Bind` function is greater than one, then the endpoint may handle simultaneous incoming connection requests. The sequence number returned by `Listen` is used to distinguish between them.

The sequence of events for accepting an incoming connection in synchronous mode is shown below.

| **Local** | **Remote** |
|---|---|
| Client makes a `Bind` call with a `qlen` greater than zero. | |
| | Remote end uses `Connect` to send a connection request. |
| The client's notifier is called with a T_LISTEN event. | |
| Client makes a `Listen` call. | |
| Client makes an `Accept` call to accept the request, or `SndDisconnect` to reject it. | |

| | |
|---|---|
| | Remote end's `Connect` function returns to the caller. If the connection was rejected, then the remote end will also issue a `RcvDisconnect` to clear the T_DISCONNECT event. |

The sequence of events for accepting an incoming connection in asynchronous mode is shown below.

| **Local** | **Remote** |
|---|---|
| Client makes a `Bind` call with a `qlen` greater than zero. The local end is now ready to receive incoming connection requests. | |
| | Remote end uses `Connect` to send a connection request. |
| The client's notifier is called with a T_LISTEN event. | |
| Client makes a `Listen` call. (Listens are never asynchronous.) | |
| Client makes an `Accept` call to accept the request, or `SndDisconnect` to reject it. | |
| | Remote end's notification routine is called with T_CONNECT to indicate the `Connect` call has completed. If the connection was rejected, then the remote end will also issue a `RcvDisconnect` to clear the T_DISCONNECT event. |
| The client's notification routine is called with either T_ACCEPTCOMPLETE or T_DISCONNECTCOMPLETE. | |

# Tearing Down a Connection

There are two ways of tearing down a connection. All connection-oriented endpoints support an abortive disconnect, and some connection-oriented endpoints may support an orderly disconnect.

In an abortive disconnect, the connection is torn down when the client makes the `SndDisconnect` function. This kind of disconnect can be a problem since data being sent is typically buffered locally before being sent. If a client makes a `Snd` call followed by a `SndDisconnect` call, the client can not be sure that the data was actually sent without agreeing upon some kind of handshake mechanism with its remote partner.






Some protocols, such as TCP, support an over-the-wire handshake when tearing down a connection. With these kinds of protocols, a client can make a `SndOrderlyDisconnect` call to initiate an orderly teardown of the connection. The underlying protocol will perform the end-to-end disconnect, and then notify the client (with a T_ORDREL event) when all buffered data has been sent (in both directions) and the connection has been torn down.

Most connection-oriented stream protocol definitions do not contain an over-the-wire mechanism for orderly disconnect (NetBIOS, ADSP, ISO TP4). However, these protocol implementations my still support the `SndOrderlyDisconnect` call. In these cases, the orderly disconnect is implemented locally. When a client issues a `SndOrderlyDisconnect`, the underlying protocol will ensure that all buffered data has been sent (and acknowledged if the protocol supports it). Only at this point, does the protocol actually tear down the connection and notify the client (with a T_ORDREL event).

## Abortive Disconnect

The sequence of events for an abortive disconnect is shown below:

| **Local** | **Remote** |
|---|---|
| Client issues `SndDisconnect` | |
| | Client receives a T_DISCONNECT event. Client makes a `RcvDisconnect` call and the endpoint state goes to T_IDLE. |
| Client's notification routine is called with a T_DISCONNECTCOMPLETE event. (asynchronous mode only) | |

**Note:** It is possible that a client may issue a `SndDisconnect` just as its remote partner is also tearing down the connection. One of the clients may receive a `kOTLookErr` result code from the `SndDisconnect` call, in which case it should issue a `RcvDisconnect` call to clear the event.

## Orderly Disconnect

The sequence of events for an abortive disconnect is shown below for protocols (like TCP) that support over-the-wire orderly disconnects.

| **Local** | **Remote** |
|---|---|
| Client issues `SndOrderlyDisconnect`. Endpoint state goes to T_OUTREL. | |
| | Client receives a T_ORDREL event. Client may continue to send and receive data. |
| Client may continue to receive data if more arrives, but client may not send any more data. | |

Client receives unread data, and then client acknowledges the T_ORDREL event by making the `RcvOrderlyDisconnect` call. The endpoint goes to the T_INREL state. The client may still send data.

Client continues to read data if remote end continues to send.

The client issues the `SndOrderlyDisconnect` call. At this point, the endpoint state goes to T_IDLE.

the connection is broken at this point

Client receives a T_ORDREL event. Client may neither receive nor send more data. Client issues a `RcvOrderlyDisconnect` to clear the event, and the endpoint state goes to T_IDLE.

The sequence of events for an abortive disconnect is shown below for protocols that do not support over-the-wire orderly disconnects (NetBIOS, ADSP, ISO TP4) but do implement orderly disconnects locally.

| **Local** | **Remote** |
|---|---|
| Client issues `SndOrderlyDisconnect`. The underlying protocol sends all buffered data and then tears down the connection. | |
| the connection is broken at this point | |
| | Client receives a T_ORDREL event. Client may continue to receive any locally buffered, unread data. |
| Client may continue to receive data (any unread data that was locally buffered before the client issued the `SndOrderlyDisconnect`), but may not send more data. | |
| | Client receives unread data, and then client acknowledges the T_ORDREL event by making the `RcvOrderlyDisconnect` call. The endpoint goes to the T_INREL state. |






The client issues the `SndOrderlyDisconnect` call. At this point, the endpoint state goes to T_IDLE.

Client receives a T_ORDREL event. Client may neither receive nor send more data. Client issues a `RcvOrderlyDisconnect` to clear the event, and the endpoint state goes to T_IDLE.

The sections that follow describe the functions for creating and tearing down connections between connection-oriented endpoints.

# Connect

**FUNCTION**

`Connect` Initiate a connection.

**C INTERFACE**

```
OSStatus    OTConnect(EndpointRef ref, TCall* sndCall, TCall* rcvCall)
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Connect(TCall* sndCall, TCall* rcvCall)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `sndCall->addr.maxlen` | x | / |
| `sndCall->addr.len` | / | / |
| `sndCall->addr.buf` | (x) | / |
| `sndCall->opt.maxlen` | x | / |
| `sndCall->opt.len` | x | / |
| `sndCall->opt.buf` | (x) | / |
| `sndCall->udata.maxlen` | / | / |
| `sndCall->udata.len` | x | / |
| `sndCall->udata.buf` | (?) | / |
| `sndCall->sequence` | / | / |
| `rcvCall->addr.maxlen` | x | / |
| `rcvCall->addr.len` | / | x |
| `rcvCall->addr.buf` | ? | (?) |
| `rcvCall->opt.maxlen` | x | / |
| `rcvCall->opt.len` | / | x |
| `rcvCall->opt.buf` | ? | (?) |
| `rcvCall->udata.maxlen` | x | / |
| `rcvCall->udata.len` | / | x |
| `rcvCall->udata.buf` | ? | (?) |
| `rcvCall->sequence` | / | / |

`Connect` lets a client of an connection-oriented endpoint request a connection to the specified remote endpoint This function can be issued only in the T_IDLE state. The `sndCall` and `recycle` parameters point to `TCall` structures, which contain the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The `sndCall` parameter specifies information needed by the endpoint to establish a connection. The `rcvCall` parameter will be filled in with information associated with the newly established connection (synchronous mode only). In asynchronous mode, the `rcvCall` parameter is ignored.

The `sndCall->addr` member specifies the protocol address of the remote endpoint.






The `sndCall->opt` member provides any protocol-specific options that the client may want to specify (See the "Option Negotiation" section). The client may choose not to negotiate protocol options by setting the `sndCall->opt.len` field to zero.

The `sndCall->udata` member contains any optional user data that may be passed to the remote endpoint during connection establishment.

The `sndCall->udata.buf` field points to a buffer containing the data, whose length is `sndCall->udata.len`.

The client may not send more data than the endpoint allows. This information is returned in the `connect` field of a `TEndpointInfo` structure filled out by the `OpenEndpoint` or `GetEndpointInfo` functions. If the sndCall`->udata.len` field is zero, no data will be sent to the remote endpoint.

The `sndCall->sequence` member has no meaning for this function.

If the endpoint is in synchronous mode, the `Connect` call will wait for the connection to be established before returning, and the `addr`, `opt`, and `udata` fields of the `TCall` structure pointed to by `rcvCall` will be updated with values associated with the connection. When a synchronous `Connect` call is interrupted because of an asynchronous event, such as a rejected connection, the state of the endpoint is set to T_OUTCON, allowing a client to call `RcvConnect` to wait for the connect to complete. Call `RcvDisconnect` to read the result of a rejected connection request.

In asynchronous mode, the `Connect` function will return after initiating the connection request before the connect function has completed. The `kOTNoDataErr` error is returned to indicate the connect is in progress. The client will receive a T_CONNECT event when the connect operation completes successfully, and must issue the `RcvConnect` function to read the connection parameters that would have been returned in the `rcvCall` structure if the `Connect` call had been issued in synchronous mode.

If the Connect function returns a result other than `kOTNoDataErr`, then the connection attempt has not been initiated and no events will be received.

When a connection is rejected, the client receives a T_DISCONNECT event. Then client must then call `RcvDisconnect` to clear the error.

**VALID STATES**

T_IDLE

**RESULT CODES**

`kOTAccessErr`
`kOTAddressBusyErr`
`kOTBadAddressErr`
`kOTBadDataErr`
`kOTBadOptionErr`
`kOTBufferOverflowErr`
`kOTCanceledErr`
`kOTLookErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

None

# RcvConnect

**FUNCTION**

`RcvConnect` Read the status of an outstanding or completed asynchronous call to the function `Connect`.

**C INTERFACE**

```
OSStatus    OTRcvConnect(EndpointRef ref, TCall* call);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvConnect(TCall* call);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `call->addr.maxlen` | x | / |
| `call->addr.len` | / | x |
| `call->addr.buf` | ? | (?) |
| `call->opt.maxlen` | x | / |
| `call->opt.len` | / | x |
| `call->opt.buf` | ? | (?) |
| `call->udata.maxlen` | x | / |
| `call->udata.len` | / | x |
| `call->udata.buf` | ? | (?) |
| `call->sequence` | / | / |

`RcvConnect` is used by a client of a connection-oriented endpoint to read the status of a previously issued `Connect`. The `Connect` call may still be pending in which case `kOTNoDataErr` is returned. The `call` parameter points to a `TCall` structure, which is filled in by the endpoint with information describing the established connection. The `TCall` structure has the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The `addr` member returns the protocol address of the endpoint that accepted the connection request. Note that this may not be the same address that received the connection request. The `addr.maxlen` field must be initialized by the client with a value large enough to hold the address before making the call.

The `opt` member is filled in with protocol-specific parameters associated with the established connection. The `opt.maxlen` field must be initialized by the client before the call with a value large enough to hold the options.

The `udata` member is filled in with data associated with the connection request.

The `udata.buf` field points to a client-supplied buffer of size `udata.maxlen` to hold the data.

The `sequence` member has no meaning for this function.






The `call` parameter may be `NULL`, in which case no information is returned to the client by `RcvConnect`.

If the endpoint is synchronous and blocking, `RcvConnect` will wait for the connection to be accepted or rejected. A `kOTNoError` result code will indicate that the connection was accepted, and a `kOTLookErr` result code will indicate that the connection was rejected (The client will need to call `Look` to verify that a `T_DISCON` event is the reason for the `kOTLookErr`, and then call `RcvDisconnect` to clear the event indication.

Otherwise, `RcvConnect` will return with a `kOTNoDataErr` if the connection attempt has not yet completed.

**VALID STATES**

T_OUTCON

**RESULT CODES**

`kOTCanceledErr`
`kOTNotSupportedErr`
`kOTOutStateErr`
`kOTNoDataErr`
`kOTBufferOverflowErr`

**SEE ALSO**

`Connect`

# Listen

**FUNCTION**

`Listen` — `Listen` for an incoming connection request.

**C INTERFACE**

```
OSStatus    OTListen(EndpointRef ref, TCall* call);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Listen(TCall* call);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `call->addr.maxlen` | x | / |
| `call->addr.len` | / | x |
| `call->addr.buf` | ? | (?) |
| `call->opt.maxlen` | x | / |
| `call->opt.len` | / | x |
| `call->opt.buf` | ? | (?) |
| `call->udata.maxlen` | x | / |
| `call->udata.len` | / | x |
| `call->udata.buf` | ? | (?) |
| `call->sequence` | / | x |

`Listen` is used by a client of a connection-oriented endpoint to listen for an incoming connection request. The `call` parameter points to a `TCall` structure which is filled in by the endpoint with information describing the connection indication. The `TCall` structure has the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The `addr` member returns the protocol address of the calling endpoint. This address is in a format usable in future calls to `Connect` or `Accept`. The `addr.maxlen` field must be initialized by the client before the call with a value large enough to hold the address.

The `opt` member is filled in with protocol-specific parameters associated with the connection request. The `opt.maxlen` field must be initialized by the client before the call with a value large enough to hold the options.

The `udata` member is filled in with data associated with the connection request.

The `udata.buf` field points to a client-supplied buffer of size `udata.maxlen` to hold the data.

The `sequence` member is filled in with a value that uniquely identifies the connect indication. Because connection indications are uniquely identified, a client can listen for multiple connect indications before responding to any of them.

If the endpoint is in synchronous mode and is blocking, `Listen` will not return until a connection indication has been received.






If the endpoint is in asynchronous mode or is not blocking, `Listen` will return any pending connection request, or it will return `kOTNoDataErr` if there are no pending connection requests.

**VALID STATES**

T_IDLE, T_INCON

**RESULT CODES**

`kOTBadQLenErr`
`kOTCanceledErr`
`kOTLookErr`
`kOTNoDataErr`
`kOTNotSupportedErr`
`kOTOutStateErr`
`kOTQFullErr`

**SEE ALSO**

None

# Accept

**FUNCTION**

Accept — Accept an incoming connection request.

**C INTERFACE**

```
OSStatus    OTAccept(EndpointRef ref, EndpointRef resRef, TCall* call);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::Accept(EndpointRef resRef, TCall* call);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| call->addr.maxlen | / | / |
| call->addr.len | x | / |
| call->addr.buf | (?) | / |
| call->opt.maxlen | / | / |
| call->opt.len | x | / |
| call->opt.buf | (?) | / |
| call->udata.maxlen | / | / |
| call->udata.len | x | / |
| call->udata.buf | (?) | / |
| call->sequence | x | / |

Accept is used by a client to accept a connection request. The client can either accept the connection on the same endpoint that received the connection request, or the client can specify another endpoint that should accept the connection. The call parameter points to a TCall structure containing the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence     sequence;
```

The addr member contains the protocol address of the calling endpoint. The client need not specify an address by setting the addr.len field to zero. If an address is provided, it may be optionally checked by the endpoint.

The opt member indicates any protocol-specific parameters associated with the connection. The values of parameters specified by opt and the syntax of those are protocol-specific. See the section on option negotiation for further discussion. If the user does not indicate any protocol-specific options (by setting opt.len to zero), it is assumed that the connection is to be accepted unconditionally. The endpoint may choose options other than the defaults to ensure that the connection is accepted successfully.

The udata member contains any user data to be returned to the calling endpoint. The amount of user data must not exceed the limits supported by the endpoint as returned in the connect field of the TEndpointInfo structure filled out on a call to OpenEndpoint or GetEndpointInfo.

The data to be sent is pointed to by udata.buf, and the number of bytes is specified by udata.len.






The sequence member is the value returned by Listen that uniquely associates the response with a previously received connect indication.

If a client accepts a connection on the same endpoint that received the connection indication, the client must have responded to all previous connect indications received on the endpoint via the Accept or SndDisconnect functions. Otherwise, the Accept function will fail with the result code kOTIndOutErr.

If a different endpoint is specified, then the client may or may not choose to bind the endpoint before the Accept call is issued. If the endpoint is not bound before the Accept call, then the transport provider will automatically bind it to the same protocol address the endpoint that received the connection request was bound to. If the client chooses to bind the endpoint, it must be bound to a protocol address with a qlen of zero and must be in the T_IDLE state before the Accept call is made. The endpoint that ends up with the open connection will receive a T_PASSCON event to indicate that a connection is open.

The call to Accept will fail with kOTLookErr if there are indications (T_DISCONNECT or T_LISTEN) waiting to be received.

**WARNING**: Calling Accept on an endpoint that was bound with a qlen greater than 1 can result in a kOTLookErr being returned because another T_LISTEN event has arrived. Unfortunately, XTI specifies that the Accept cannot be acted on until a Listen has been issued to receive this new connection request. This effectively means that you need to keep an array of outstanding connection requests. If you are acting on T_LISTEN events in your notifier, then you need to be able to handle having "qlen" outstanding connection requests, issuing an Accept, and getting a T_LISTEN event before the Accept returns to you.

If the endpoint is in asynchronous mode the Accept function will return immediately. A result code of kOTNoError indicates that the Accept has begun and the client will be notified when it is complete.

The endpoint that issued the accept will receive a T_ACCEPTCOMPLETE event. The endpoint receiving the connection will receive a T_PASSCON event. In the case where these two endpoints are the same, the endpoint will receive both events. The cookie parameter for the T_ACCEPTCOMPLETE event is the EndpointRef of the endpoint that issued the accept. The cookie paramter for the T_PASSCON event is the EndpointRef of the endpoint that received the connection.

If the Accept fails, the connection indication is still outstanding, and still needs to be dealt with (probably by issuing a SndDisconnect).

If a notification routine is not installed, the client can poll the accepting endpoint, waiting for the state to change to T_DATAXFER

**Note:** In asynchronous mode, it is possible for the endpoint to issue the T_ACCEPTCOMPLETE event before the Accept function returns the kOTNoError result to the client.

**VALID STATES**

ref (C) or this: (C++) T_INCON

resRef T_IDLE or T_UNBND

**RESULT CODES**

kOTBadAddressErr

kOTBadDataErr

kOTBadOptionErr

kOTBadReferenceErr

kOTBadSequenceErr

kOTCanceledErr
kOTIndOutErr
kOTLookErr
kOTNotSupportedErr
kOTOutStateErr
kOTProviderMismatchErr
kEPROTOErr

**SEE ALSO**

SndDisconnect , Listen






# SndDisconnect

**FUNCTION**

SndDisconnect — Tear down an open connection or refuse an incoming connection request.

**C INTERFACE**

```
OSStatus    OTSndDisconnect(EndpointRef ref, TCall* call);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndDisconnect(TCall* call);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| call->addr.maxlen | / | / |
| call->addr.len | / | / |
| call->addr.buf | / | / |
| call->opt.maxlen | / | / |
| call->opt.len | / | / |
| call->opt.buf | / | / |
| call->udata.maxlen | / | / |
| call->udata.len | x | / |
| call->udata.buf | (?) | / |
| call->sequence | ? | / |

SndDisconnect initiates an abortive release on an already established connection, or rejects a connection request. The call parameter points to a TCall structure, which contains the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The values in this structure have different semantics depending upon the context of the call to SndDisconnect. When rejecting a connection request, call must be non-NULL and contain a valid value for sequence to uniquely identify the rejected connect indication to the endpoint. The sequence field is meaningful only if the connection is in the T_INCON state. The addr and opt fields of call are ignored. In all other cases, call need only be used when data is being sent with the disconnect request. The addr, opt, and sequence fields of the TCall structure are ignored. If the client does not wish to send data to the remote client, the value of call may be a null pointer.

The udata member specifies the client data to be sent to the remote client. The amount of user data must not exceed the limits supported by the endpoint, as returned in the discon field of the TEndpointInfo structure filled out by the OpenEndpoint or GetEndpointInfo functions. The data to be sent is pointed to by udata.buf, and the number of bytes is specified by udata.len.

If the endpoint is in asynchronous mode the SndDisconnect function will return immediately. A result code of kOTNoError indicates that the SndDisconnect has begun and the client will be notified when it is complete.

When the disconnect has been completed and a notification routine has been installed (See InstallNotifier), a T_DISCONNECTCOMPLETE event will be issued.

If a notification routine has not been installed, it is not possible to determine when the SndDisconnect function is complete.

The cookie parameter passed to the notification routine to indicate completion is the call parameter.

**VALID STATES**

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL (and T_INCON, when two or more incoming connection requests are outstanding)

**RESULT CODES**

kOTBadDataErr
kOTBadSequenceErr
kOTCanceledErr
kOTLookErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

None






# RcvDisconnect

**FUNCTION**

| | |
|---|---|
| RcvDisconnect | Identify the cause of a disconnect, and acknowledge the corresponding disconnect event. |

**C INTERFACE**

```
OSStatus    OTRcvDisconnect(EndpointRef ref, TDiscon* discon);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvDisconnect(TDiscon* discon);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| discon->udata.maxlen | x | / |
| discon->udata.len | / | x |
| discon->udata.buf | ? | (?) |
| discon->reason | / | x |
| discon->sequence | / | ? |

RcvDisconnect identifies the reason that a connection that was torn down or that a connection request failed or was rejected. RcvDisconnect clears the corresponding disconnect event, and retrieves any user data sent with the disconnect. The client passes a pointer to a TDiscon structure, in which the endpoint returns the reason for the disconnect and returns any data sent with the disconnect. The TDiscon structure has the following members:

```
struct TNetbuf udata;
OTReason reason;
OTSequence sequence;
```

The reason field specifies the reason for the disconnect through a protocol-dependent reason code.

The udata field is filled in with any user data that was sent with the disconnect. The udata.buf field points to a client-supplied buffer of size udata.maxlen to hold the request.

The sequence field may identify an outstanding connection indication with which the disconnect is associated. The sequence field is meaningful only when RcvDisconnect is issued by a passive endpoint client that has issued one or more Listen functions and is processing the resulting connect indications. If a disconnect indication occurs, sequence can be used to identify which of the outstanding connection indications is associated with the disconnect.

If a client does not care if there is incoming data and does not need to know the value of reason or sequence, the discon parameter may be a NULL pointer. In this case, any user data associated with the disconnect will be discarded. However, if a client has retrieved more than one outstanding connect indication (via Listen), and the discon parameter is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

RcvDisconnect behaves exactly the same in all operational modes of an endpoint. If there is no disconnect pending, kOTNoDisconnectErr will be return. If there is, either kOTNoError or kOTBufferOverflowErr will be returned.

**VALID STATES**

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON (when number of outstanding incoming connection requests > 1)

**RESULT CODES**

kOTNoDisconnectErr
kOTNotSupportedErr
kOTOutStateErr
kOTBufferOverflowErr

**SEE ALSO**

None






# SndOrderlyDisconnect

**FUNCTION**

SndOrderlyDisconnect Initiate an orderly tear-down of a connection.

**C INTERFACE**

```
OSStatus    OTSndOrderlyDisconnect(EndpointRef ref);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndOrderlyDisconnect();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |

SndOrderlyDisconnect initiates an orderly release of a connection and indicates to the endpoint that the client has no more data to send. After calling SndOrderlyDisconnect, the client must not send any more data over the connection. However, the client may continue to receive data if an orderly release indication has not been received.

This function is an optional service of the endpoint; it is supported only if the endpoint returned T_COTS_ORD or T_TRANS_ORD in the servtype field of the TEndpointInfo structure filled out by the OpenEndpoint or GetEndpointInfo functions.

SndOrderlyDisconnect behaves exactly the same in all operational modes of an endpoint. The return value will be kOTNoError if the function succeeded, and an error result if it did not.

**VALID STATES**

T_DATAXFER, T_INREL

**RESULT CODES**

kOTLookErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

None

# RcvOrderlyDisconnect

**FUNCTION**

`RcvOrderlyDisconnect` Acknowledge an incoming request for an orderly connection tear-down.

**C INTERFACE**

```
OSStatus    OTRcvOrderlyDisconnect(EndpointRef ref);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvOrderlyDisconnect();
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |

`RcvOrderlyDisconnect` acknowledges receipt of an orderly release indication. After receiving this indication, the user must not attempt to receive more data; any subsequent calls to `Rcv` return the result code `kOTOutStateErr`. The user can, however, continue sending data over the connection, if `SndOrderlyDisconnect` has not yet been called by the client. This function is an optional service of the endpoint and is supported only if the endpoint returned T_COTS_ORD or T_TRANS_ORD in the `servtype` field of the `TEndpointInfo` structure filled out by the `OpenEndpoint` or `GetEndpointInfo` functions.

`RcvOrderlyDisconnect` behaves exactly the same in all operational modes of an endpoint. If there is no disconnect pending, `kOTNoReleaseErr` will be return. If there is, either `kOTNoError` or `kOTBufferOverflowErr` will be returned.

**VALID STATES**

T_DATAXFER, T_OUTREL

**RESULT CODES**

`kOTLookErr`

`kOTNoReleaseErr`

`kOTNotSupportedErr`

`kOTOutStateErr`

**SEE ALSO**

None






# Using Connection-Oriented Streams

Open Transport supports connection-oriented stream endpoints. There are two Open Transport functions specific to this type of endpoint—`Snd` and `Rcv`.

A client uses the `Snd` function to send data to the remote endpoint and uses `Rcv` to receive data from the remote endpoint. Some endpoints support expedited data in addition to normal data. However, protocol endpoints that support expedited data usually handle the data in significantly different ways depending upon the particular protocol being implemented. In general, clients should not use expedited data as this will lead to non-transport-independent code.

Some endpoints support the concept of logical separators in the data stream that will be passed from one endpoint to another. These logical separators break the stream up into Transport Service Data Units (TSDU) Some endpoints have a maximum size TSDU that may be supported, while other endpoints have no size limit. A client uses the `GetEndpointInfo` function to find out what the TSDU size is for both normal and expedited data (the ETSDU). The values can be found in the `tsdu` and `etsdu` fields of the `TEndpointInfo` structure. A value of 0 indicates that no logical separator is supported, a value of -1 indicates that there is no maximum size to a TSDU, a positive number indicates the maximum size of a TSDU, and -2 indicates that normal (or expedited in the case of `etsdu`) data is not supported by the endpoint.

**Note**: Some endpoints that support TSDUs have a variable maximum size limit. A client should be aware that the values for `tsdu` and `etsdu` that are returned when the endpoint is opened (with `OTOpenEndpoint`) may be different if queried again using `GetEndpointInfo` after a connection has been opened because the endpoints may have negotiated different values during the connection-establishment process.

# Snd

**FUNCTION**

Snd — Send data on a connection-oriented stream.

**C INTERFACE**

```
OTResult    OTSnd(EndpointRef ref, void* buf, size_t nbytes, OTFlags
            flags);
```

**C++ INTERFACE**

```
OTResult    TEndpoint::Snd(void* buf, size_t nbytes, OTFlags flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| buf | (x) | / |
| nbytes | x | / |
| flags | x | / |

Snd is used by the client of a connection-oriented stream protocol to send either normal or expedited data. On successful completion, Snd returns an integer value of zero or greater, indicating the number of bytes sent. On error, Snd returns a negative integer corresponding to a result code.

To specify the data to be sent, the client passes a pointer to data and a length. The client may specify any optional flags in the flags parameter:

T_EXPEDITED — If set, the data will be sent as expedited data (if supported by the endpoint)

T_MORE — If set, this indicates that the transport service data unit (TSDU) or expedited transport service data unit, ETSDU) is being sent with multiple Snd calls. Each Snd with the T_MORE flag set indicates that another Snd will follow with more data for the current TSDU.

The end of the TSDU (or ETSDU) is identified with a Snd call with the T_MORE flag not set. Use of T_MORE allows the client to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the endpoint. If the endpoint does not support the concept of TSDU as indicated in the info field of a TEndpointInfo structure filled out by either the OpenEndpoint or GetEndpointInfo functions, the T_MORE flag is not meaningful and will be ignored if set.






The sending of a zero length fragment of a TSDU or ETSDU is permitted only where this is used to indicate the end of a TSDU or ETSDU, that is when the T_MORE flag is not set. Some endpoints forbid the zero length TSDUs and ETSDUs. In this case a kOTBadDataErr error will result.

If the endpoint is in non-blocking or asynchronous mode, it is possible that only part of the data will actually be accepted by the transport provider. In this case, *OTSnd* will return a value that is less than the value of the nbytes parameter, or the error *kOTFlowErr* if no bytes at all were sent. After this error occurs, a T_GODATA event will be issued when the flow control restrictions are lifted. This error will never be returned if the endpoint is in blocking mode.

If an asynchronous event, such as a disconnect event, occurs which interrupts the Snd function, it will return with the kOTLookErr result.

The behavior of Snd is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | kOTFlowErr never returned<br>Returns when flow control lifts |
| **Sync/Non-Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately |
| **Async/Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately |
| **Async/Non-Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately |

The client may negotiate the XTI_SNDLOWAT option with an endpoint using the OptionManagement function. This option value gives the minimal number of bytes that must have accumulated in the endpoint's send buffer before they are sent. Not all endpoints support the XTI_RCVLOWAT option.

If the function fails with the result code kOTBadDataErr, one of the following conditions occurred:

- A single send was attempted specifying a TSDU(ETSDU) or fragment TSDU(ETSDU) greater than that specified by the current values for TSDU or ETSDU for this endpoint.
- A send of a zero-byte TSDU(ETSDU) or zero byte fragment of a TSDU(ETSDU) is not supported by this endpoint.
- Multiple sends were attempted, resulting in a TSDU(ETSDU) larger than that specified by the current values of TSDU or ETSDU for this endpoint.

**VALID STATES**

T_DATAXFER, T_INREL

**RESULT CODES**

kOTBadDataErr

kOTBadFlagErr
kOTCanceledErr
kOTFlowErr
kOTLookErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

AckSends, GetEndpointInfo, Rcv






# Rcv

**FUNCTION**

Rcv — Read data on a connection-oriented stream.

**C INTERFACE**

```
OTResult    OTRcv(EndpointRef ref, void* buf, size_t nbytes, OTFlags*
            flags);
```

**C++ INTERFACE**

```
OTResult    TEndpoint::Rcv(void* buf, size_t nbytes, OTFlags* flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| buf | x | (x) |
| nbytes | x | / |
| flags | x | (x) |

Rcv receives either normal or expedited data. On successful completion, Rcv returns an integer value of zero or greater, indicating the number of bytes received. On error, Rcv returns a negative integer corresponding to a result code.

The client must specify an area in memory to which the data should be copied.

On return, if T_MORE is set in the flags, this indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple Rcv calls. In asynchronous mode, T_MORE may be set on return even when the number of bytes received is less than the size of the receive buffer specified. Each Rcv with the T_MORE flag set indicates that another Rcv must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a Rcv call with the T_MORE flag not set. If the endpoint does not support the concept of a TSDU, the T_MORE flag is not meaningful and should be ignored. If the client requests more than zero bytes on the call to Rcv, then the function will return zero only if the end of a TSDU is being returned to the client.

On return, the data is expedited data if T_EXPEDITED is set in flags. If the number of bytes of expedited data exceeds the number of bytes requested in reqCount, the T_EXPEDITED and T_MORE flags will both be set. Subsequent calls to Rcv to will return the remaining ETSDU.

**Note:** If the client is in the middle of reading normal data TSDU, and then a Rcv returns expedited data, the next Rcv that returns without the T_EXPEDITED flag will return normal data at the place it was interrupted. It is the responsibility of the client to remember their place in the normal data stream when interrupted by expedited data.

The reqCount parameter is filled in with the actual number of bytes read.

If the client has installed a notification routine , the T_DATA or T_EXDATA events will be issued when there is data available. If no notification routine is installed, the client may poll for these events by repeatedly calling the Look function.

If the endpoint is in synchronous mode and is blocking, the endpoint will wait for data if none is currently available. Generally, this method of operation is discouraged as it may lead to a 'hang' if no data ever becomes available. If the client is doing other operations in synchronous mode, it should put the endpoint in non-blocking mode before making the `Rcv` call.

If the endpoint is in asynchronous mode or is not blocking, the function will fail with the `kOTNoDataErr` result if no data is available. Once a client gets the T_DATA event, it should continue in a loop making the `RcvUData` call until a `kOTNoDataErr` error is returned.

Clients should be prepared for a T_DATA event and then a `kOTNoDataErr` error when a `Rcv` call is made. This seems unusual, but it can occur if you are calling Rcv in the foreground when a T_DATA event comes in.

One other situation that is worth noting is that a Rcv can get a kOTLookErr error returned from the call. It is VERY important that you actually do the OTLook. If you are in a flow-control situation on the send side, and a T_GODATA or T_GOEXDATA event occurs that you do not clear in your notifier (by calling OTLook or by actually sending some data), then if you do not do the OTLook in response to a kOTLookErr error from a Rcv call, you will hang waiting for events. Until the T_GODATA or T_GOEXDATA are cleared, Open Transport cannot send you another T_DATA event (or any other event other than a T_DISCONNECT, for that matter).

The client may negotiate the XTI_RCVLOWAT option with an endpoint using the `OptionManagement` function. This option value gives the minimal number of bytes that must have accumulated in the endpoint's receive buffer before a T_DATA event is issued. Not all endpoints support the XTI_RCVLOWAT option.

**VALID STATES**

T_DATAXFER, T_OUTREL

**RESULT CODES**

`kOTCanceledErr`
`kOTLookErr`
`kOTNoDataErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

`Look`, `SetAsynchronous`, `SetBlocking`, `Snd`






# Processing Transactions

This section describes how a client uses the Open Transport transaction protocol calls to transfer data between two endpoints. A client can either actively initiate a transaction request, or it can passively wait for an endpoint to receive an incoming transaction request. Each transaction is assigned a unique sequence number to identify it for issuing/receiving the reply. There can be any number of transactions outstanding at the same time.

## Initiating a Transaction Request

In order to initiate a transaction request on an endpoint, the client must first bind the endpoint with the `Bind` function.

The client uses the `SndURequest` function to initiate a transaction. The parameters to the call include the address of the remote end, any data to send along with the transaction request (if the particular protocol the endpoint implements allows it), and any options that the client wants to specify.

Transactions come in two types: acknowledged and unacknowledged. The default is unacknowledged. Set the T_ACKNOWLEDGED bit in the OTFlags parameter of the SndURequest to get acknowledged transactions. From the client's perspective the only difference between the two is that when a reply is sent to an incoming acknowledged request, the SndUReply function does not complete until the reply is acknowledged. In this case, kOTNoError will be returned if the reply was acknowledged, and kETIMEDOUTErr will be returned if the transaction protocol timed out waiting for acknowledgment. For unacknowledged transactions, the SndUReply function completes immediately.

There is no difference between issuing requests and receiving responses in synchronous or asynchronous modes.

The sequence below shows the order of events for a successful transaction request.

| **Local** | **Remote** |
|---|---|
| Client makes a `SndURequest` call, which returns immediately with no error. | |
| | The remote client's notification routine is called with a T_REQUEST event. |
| | Remote end issues a `RcvURequest`, formulates a response, issues a `SndUReply` with the requested data (and sequence # identifier) |
| Client receives a `T_REPLY` event and issues `RcvUReply` to receive the reply. `kOTNoError` is returned and the `TUnitReply` structure contains the reply information. | |

If remote end rejects the transaction request, or the request fails in some other way, this is the sequence of events:

| Local | Remote |
|---|---|
| Client makes a `SndURequest` call, which returns immediately with no error. | |
| | The remote client's notification routine is called with a T_REQUEST event. |
| | Remote end rejects the transaction request. |
| The protocol will retry the request to protocol specification. | |
| Client receives a `T_REPLY` event and issues `RcvUReply` to receive the reply. A `kETIMEDOUTErr` is returned, and the `TUnitReply` structure contains no useful information other than the sequence number of the corresponding SndURequest. | |

# Responding to a Transaction Request

In order to respond to a transaction request on an endpoint, the client must first bind the endpoint with the `Bind` function.

Synchronous clients in blocking mode may issue a `RcvURequest` call to wait for an incoming request.

Clients will receive a T_REQUEST event notification when a transaction request arrives (unless a RcvURequest call is in progress by a synchronous, blocking client). The client must issue the `RcvURequest` function to retrieve the information associated with the transaction request. This information includes the remote address, any options associated with the request, any data associated with the request, and a sequence number.

The client can either reject the incoming transaction request by calling `CancelUReply`, or the client can accept the incoming transaction request, formulate an appropriate response, and reply by calling the `SndUReply` function and passing the sequence number returned from the `RcvURequest` function to indicate which transaction request the reply is for.






## Synchronous/Blocking Response

The sequence of events for handling an incoming transaction request in synchronous mode is shown below

| Local | Remote |
|---|---|
| Client makes a `Bind` call. | |
| Client makes a `RcvURequest` call. | |
| | Remote end uses `SndURequest` to send a transaction request. |
| `RcvURequest` returns to client with request info and sequence #. | |
| Client makes an `SndUReply` call to respond to the request. If this was an acknowledged request, the SndUReply will complete with either kOTNoError if the reply was acknowledged, or kETIMEDOUTErr if it was not. | |
| | Remote end's notification routine is called with T_REPLY to indicate that reply data is available. `RcvUReply` is called to receive the reply data. |

### Asynchronous Response

The sequence of events for accepting an incoming transaction request in asynchronous mode is shown below

| Local | Remote |
|---|---|
| Client makes a `Bind` call. The local end is now ready to receive incoming transaction requests. | |
| | Remote end uses `SndURequest` to send a transaction request. |
| The client's notification routine is called with a T_REQUEST event. | |
| Client makes a `RcvURequest` call to obtain request info and sequence #. | |
| Client makes an `SndUReply` call to respond to the request. | |
| | Remote end's notification routine is called with T_REPLY to indicate the `SndURequest` call has completed. |
| The client's notification routine is called with T_REPLYCOMPLETE and the cookie containing the sequence value generated by the client for the corresponding request. The result code will be either kOTNoError if the reply was acknowledged, or kETIMEDOUTErr if it was not | |

## Connectionless Transactions

Open Transport supports connectionless transaction endpoints. The Open Transport functions specific to this type of endpoint are `SndURequest`, `RcvURequest`, `SndUReply, and RcvUReply`.

`SndURequest` is used by a client to initiate a transaction; the `SndURequest` function must have a unique (among all currently outstanding outgoing requests) non-zero sequence number value filled out in the TUnitReply structure. The sequence number is required, since multiple requests may be outstanding at any time, and replies may not arrive in the order that the requests were issued. The `RcvUReply` function is used to read incoming replies. Since incoming replies do not necessarily arrive in the same order as the requests were sent, it is necessary to be prepared to receive a reply to any outstanding requests. One method for dealing with this is to call `RcvUReply` with no data buffer. This will result in the sequence number being stored in the `TUnitReply` structure, and the `T_MORE` flag being set in the `OTFlags` parameter (unless an error occurred). Once a reply has been partially received with the T_MORE flag being set, it is guaranteed that subsequent calls to `RcvUReply` will read from the same reply, until the function returns with the `T_MORE` flag cleared.






The endpoint that is the destination of a transaction request reads the request using the `RcvURequest` function. A sequence number is returned along with the request data; the responding client must return this sequence number along with the response data when making the `SndUReply` function so that the endpoint can determine which request the client is responding to. This sequence number is required because one endpoint (depending upon the implementation) may have more than one concurrent transaction outstanding at any given time. A responding client need not respond to requests in the order received.

# SndURequest

**FUNCTION**

SndURequest — Send a unit request.

**C INTERFACE**

```
OSStatus    OTSndURequest(EndpointRef ref, TUnitRequest* req, OTFlags
            reqFlags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndURequest(TUnitRequest* req, OTFlags reqFlags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| req->addr.maxlen | / | / |
| req->addr.len | x >= 0 | / |
| req->addr.buf | (x) | / |
| req->opt.maxlen | / | / |
| req->opt.len | x >= 0 | / |
| req->opt.buf | (?) | / |
| req->udata.maxlen | / | / |
| req->udata.len | x >= 0 | / |
| req->udata.buf | (x) | / |
| req->sequence | x | / |
| reqFlags | x | / |

SndURequest is used by the client of a connectionless transaction endpoint to send a unit request to another endpoint. The client passes a TUnitRequest structure and reqFlags that contain the remote address, the request data, any options, and flags.

The TUnitRequest structure has the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The addr member contains the protocol address of the destination of the request. The addr.len field holds the length of the protocol address.

The opt member contains any protocol-specific options for the request. The opt.len field holds the length of the options, or zero if there are none.

The udata member contains the request data. The udata.len field specifies the length of the request. This value must not exceed the one returned in the etsu field of the TEndpointInfo structure filled in by OpenEndpoint or GetEndpointInfo.

A client may send non-contiguous data by setting the req->udata.buf pointer to point to an OTData structure, and setting the req->udata.len value to kNetbufDataIsOTData

The sequence field should be set to a non-zero value that will uniquely identify this request from all other outstanding requests on the endpoint.






The only defined flag for SndURequest are the T_ACKNOWLEDGED and T_MORE flags. The T_MORE flag indicates that more request data will be sent in a subsequent SndURequest call. The T_ACKNOWLEDGED flag indicates that the request is required to be acknowledged. This flag may not be honored by all transaction protocols.

If the endpoint is in non-blocking or asynchronous mode, the SndURequest function will return a kOTFlowErr if flow control restrictions prevent the data from being accepted by the transport provider at the time the function is issued. After this error occurs, a T_GODATA event will be issued when the flow control restrictions are lifted. This error will never be returned if the endpoint is in blocking mode.

The behavior of SndURequest is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | kOTFlowErr never returned<br>Returns when flow control lifts and the request data has been sent to the protocol. |
| **Sync/Non-Blocking** | kOTFlowErr may be returned<br>Returns if flow control restrictions are in effect or the request data has been sent to the protocol. |
| **Async/Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately |
| **Async/Non-Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately |

**VALID STATES**

T_IDLE

**RESULT CODES**

kOTBadAddressErr
kOTBadFlagErr
kOTBadOptionErr
kOTCanceledErr
kOTFlowErr
kOTLookErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

RcvURequest, RcvUReply

# RcvURequest

**FUNCTION**

RcvURequest  Read an incoming unit request.

**C INTERFACE**

```
OSStatus    OTRcvURequest(EndpointRef ref, TUnitRequest* req, OTFlags*
            flags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvURequest(TUnitRequest* req, OTFlags* flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| req->addr.maxlen | x | / |
| req->addr.len | / | x |
| req->addr.buf | ? | (?) |
| req->opt.maxlen | x | / |
| req->opt.len | / | x |
| req->opt.buf | ? | (?) |
| req->udata.maxlen | x | / |
| req->udata.len | / | x |
| req->udata.buf | ? | (?) |
| req->sequence | / | x |
| flags | / | x |

RcvURequest is used by a client of a connectionless transaction endpoint to read a unit request. The client supplies a TUnitRequest structure that will be filled out with the protocol address of the originator of the request, any options associated with the request, the request data, and a sequence number to be passed back to the endpoint when a client makes a reply.

If the endpoint is in synchronous mode and is blocking, this function will wait for a request to arrive. If the endpoint is in asynchronous mode or is not blocking, the function will return any unread requests and the kOTNoDataErr result if there are no unread requests.

The endpoint will generate a T_REQUEST event when a request arrives. The client may poll for the arrival of a request by making the Look function or repeatedly calling this function for as long as the kOTNoDataErr result is returned. If the client has a notification routine installed on the endpoint, the event will be sent to the notification routine .

The TUnitRequest structure has the following members:

```
struct TNetbuf addr;
struct TNetbuf opt;
struct TNetbuf udata;
OTSequence sequence;
```

The addr member will be filled in with the protocol address of the originator of the request. The addr.maxlen field must be large enough to hold the protocol address or a kOTBufferOverflowErr error will result and the incoming request will be dropped.






The opt member contains any protocol-specific options for the request. The opt.maxlen field must be large enough to hold the options.

The udata member will be filled in with the request data.

The udata.buf field points to a client-supplied buffer of size udata.maxlen; this buffer holds the reply. To hold the largest possible reply, the udata.maxlen field should be set to a value equal to that of the tsdu field in the TEndpointInfo structure filled in by OpenEndpoint or GetEndpointInfo.

The sequence member will be filled out with the endpoint-assigned sequence number for the transaction. This sequence number must be passed by the client to the SndUReply function when sending the transaction reply, or to the CancelUReply function to cancel the transaction.

The flags parameter will be filled in with the T_MORE flag if the client-supplied buffer is not large enough to hold the entire request. The client must reissue the RcvURequest function to retrieve the rest of the request. The addr and options fields will be ignored on these subsequent calls to RcvURequest.

The flags parameter may also contain the T_ACKNOWLEDGED bit if the request has been identified as an acknowledged request rather than an unacknowledged request. This bit will only be set on the first call to RcvURequest.

In addition, the flags parameter may also contain the T_PARTIALDATA bit. In this case, the request data being received is only partial, and there is more coming, but it has not yet arrived. The difference between T_MORE and T_PARTIALDATA is that the T_MORE indicates that there is more data, and the next call to RcvURequest will read that data, while the T_PARTIALDATA flag does not have that guarantee. Like the T_ACKNOWLEDGED bit, the T_PARTIALDATA bit will only be set on the first call to RcvURequest.

**VALID STATES**

T_IDLE

**RESULT CODES**

kOTBadReferenceErr
kOTBadSyncErr
kOTCanceledErr
kOTLookErr
kOTNoDataErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

SndURequest, SndUReply, RcvUReply

# SndUReply

**FUNCTION**

`SndUReply` Send a unit reply.

**C INTERFACE**

```
OSStatus    OTSndUReply(EndpointRef ref, TUnitReply* reply, OTFlags
            flags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndUReply(TUnitReply* reply, OTFlags flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `reply->opt.maxlen` | / | / |
| `reply->opt.len` | x | / |
| `reply->opt.buf` | (x) | / |
| `reply->udata.maxlen` | / | / |
| `reply->udata.len` | x | / |
| `reply->udata.buf` | (x) | / |
| `reply->sequence` | x | / |
| `flags` | x | / |

`SndUReply` is used by the client of a connectionless transaction endpoint to send a reply in response to an incoming request. The client passes a `TUnitReply` structure containing the following members:

```
struct TNetbuf udata;
struct TNetbuf opt
OTSequence sequence;
```

The `udata` `TNetbuf` contains the reply data. The reply data is pointed to by the `udata.buf` field.

A client may send non-contiguous data by setting the reply->udata.buf pointer to point to an OTData structure, and setting the reply->udata.len value to kNetbufDataIsOTData

The `opt` `TNetbuf` contains any options that apply to the reply.

The client must pass the sequence number returned by the `RcvURequest` function in the `sequence` parameter.

The client does not need to specify the remote address; the endpoint uses the sequence to match the reply against a pending request, so the endpoint knows where to send the response.






The behavior of SndUReply is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | `kOTFlowErr` never returned<br>Returns when flow control lifts and the reply has been acknowledged or timed out(if the matching request was an acknowledged request) |
| **Sync/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately for unacknowledged requests, and when the reply has been acknowledged or timed out for acknowledged requests. |
| **Async/Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately<br>A `T_REPLYCOMPLETE` event is sent to the notification routine when the reply is acknowledged or timed out. |
| **Async/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately<br>A `T_REPLYCOMPLETE` event is sent to the notification routine when the reply is acknowledged or timed out. |

If the reply was not successfully sent (i.e. timed out - only for acknowledged requests), the `result` parameter will be set to kETIMEDOUTErr. For unacknowledged requests, a T_REPLYCOMPLETE event will still be generated for asynchronous clients so that the logic is the same for replying to both acknowledged and unacknowledged requests.

The `cookie` parameter passed to the notification routine to indicate completion will be set to the the sequence value generated by the client for the corresponding request. ( In cases where T_MORE is used to send the reply in multiple "chunks", the first `TUnitReply*` is used).

**VALID STATES**

T_IDLE

**RESULT CODES**

`kOTBadFlagErr`
`kOTBadSequenceErr`
`kOTCanceledErr`
`kOTLookErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

RcvURequest






# RcvUReply

**FUNCTION**

RcvUReply Receive a unit reply.

**C INTERFACE**

```
OSStatus    OTRcvUReply(EndpointRef ref, TUnitReply* reply, OTFlags*
            replyFlags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvUReply(TUnitReply* reply, OTFlags*
            replyFlags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| reply->opt.maxlen | x | / |
| reply->opt.len | / | x |
| reply->opt.buf | x | (x) |
| reply->udata.maxlen | x | / |
| reply->udata.len | / | x |
| reply->udata.buf | x | (x) |
| reply->sequence | / | x |
| replyFlags | x | (x) |

RcvUReply is used by the client of a connectionless transaction endpoint to receive a reply generated by a call to SndURequest. The client passes a TUnitReply structure containing the following members:

```
struct TNetbuf udata;
struct TNetbuf opt
OTSequence sequence;
```

The udata TNetbuf contains the buffer for the reply data. The reply data will be pointed to by the udata.buf field.

The opt TNetbuf contains any options that apply to the reply.

The sequence field will be filled in with the sequence number of the matching SndURequest.

The flags parameter will be filled in with the T_MORE bit set if the RcvUReply call did not read all of the reply data.

Since it is not possible to know ahead of time which request the incoming reply will match, the client must be prepared to receive a reply to any outstanding request. One way to deal with this is to first call RcvUReply with a zero value in reply->udata.maxlen. This will return the sequence number and the option information, as well as setting the T_MORE flag (unless an error occurred - then there is no data to read). Once the matching request and the appropriate reply buffer have been found, a second RcvUReply may be issued to read the actual reply data. On these second and subsequent reads, the reply->opt.len field will be set to 0. It is guaranteed that once a reply has been partially read and set the T_MORE flag, subsequent calls to RcvUReply will read from that same reply until all the data has been read.

In addition, the flags parameter may also contain the T_PARTIALDATA bit. In this case, the reply data being received is only partial, and there is more coming, but it has not yet arrived. The difference between T_MORE and T_PARTIALDATA is that the T_MORE indicates that there is more data, and the next call to RcvUReply will read that data, while the T_PARTIALDATA flag does not have that guarantee. The T_PARTIALDATA bit will only be set on the first call to RcvUReply.

Clients in asynchronous mode will receive a T_REPLY event to indicate that incoming reply data is available. Once this occurs, RcvUReply should be called repeatedly to read data until it returns a kOTNoDataErr. Otherwise, no future T_REPLY events will be received.

If a transaction has timed out awaiting reply data, the RcvUReply function will return a kETIMEDOUTErr, and the reply->sequence field will indicate which request timed out.

**VALID STATES**

T_IDLE

**RESULT CODES**

kOTBadSequenceErr
kOTCanceledErr
kOTLookErr
kOTNoDataErr
kETIMEDOUTErr

**SEE ALSO**

SndURequest






# CancelURequest

**FUNCTION**

CancelURequest Cancel an outstanding SndURequest.

**C INTERFACE**

```
OSStatus    OTCancelURequest(EndpointRef ref, OTSequence seq);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::CancelURequest(OTSequence seq);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| seq | x | . |

CancelURequest is used by the client of a connectionless transaction endpoint to cancel an outstanding outgoing request function (a call to SndURequest). Calling this function tells the protocol that this transaction is no longer of interest, and allows it to free up any memory associated with this transaction There is no acknowledgment from this function. If the sequence number indicated is not a valid sequence number, then nothing will be done.

It is the responsibility of the client to destroy any data structures associated with the canceled request.

If the value of the seq parameter is set to zero, then ALL outstanding SndURequest functions will be canceled.

Be sure to call CancelUReply if you are canceling an incoming request. This is necessary because the OTSequence value of incoming requests are generated by the protocol, while the OTSequence value of outgoing requests are generated by the client, and they may overlap.

**VALID STATES**

T_IDLE

**RESULT CODES**

kENOSRErr

**SEE ALSO**

CancelUReply

# CancelUReply

**FUNCTION**

`CancelUReply` Cancel an outstanding RcvURequest.

**C INTERFACE**

```
OSStatus    OTCancelUReply(EndpointRef ref, OTSequence seq);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::CancelUReply(OTSequence seq);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `seq` | x | . |

`CancelUReply` is used by the client of a connectionless transaction endpoint to cancel an outstanding incoming request (received by RcvURequest). Calling this function tells the protocol that this transaction is no longer of interest, and allows it to free up any memory associated with this transaction. There is no acknowledgment from this function. If the sequence number indicated is not a valid sequence number, then nothing will be done.

It is the responsibility of the client to destroy any data structures associated with the canceled request.

If the value of the `seq` parameter is set to zero, then ALL outstanding incoming requests will be canceled.

Be sure to call `CancelURequest` if you are canceling an outgoing request. This is necessary because the `OTSequence` value of incoming requests are generated by the protocol, while the `OTSequence` value of outgoing requests are generated by the client, and they may overlap.

**VALID STATES**

T_IDLE

**RESULT CODES**

`kENOSRErr`

**SEE ALSO**

`CancelURequest`

# Connection-Oriented Transactions

Open Transport supports connection-oriented transaction endpoints. The functions specific to this type of endpoint are `SndRequest`, `RcvRequest`, `SndReply`, and `CancelRequest`.






# SndRequest

**FUNCTION**

`SndRequest` Send a request.

**C INTERFACE**

```
OSStatus    OTSndRequest(EndpointRef ref, TRequest* req, OTFlags
            reqFlags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndRequest(TRequest* req, OTFlags reqFlags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->data.maxlen` | / | / |
| `req->data.len` | x | / |
| `req->data.buf` | (x) | / |
| `req->opt.maxlen` | / | / |
| `req->opt.len` | `x >= 0` | / |
| `req->opt.buf` | (x) | / |
| `req->sequence` | x | / |
| `reqFlags` | x | / |

`SndRequest` is used by the client of a connection-oriented transaction endpoint to send a request to an endpoint on the other end of the connection. The client passes a `TRequest` structure and `reqFlags` parameter that contain the request data, options, and flags.

The `TRequest` structure has the following members:

```
struct TNetbuf data;
struct TNetbuf opts
OTSequence sequence;
```

The `data` member contains the request data. If the `data.len` field is non-negative, the `data.buf` field contains a pointer to the request data.

A client may send non-contiguous data by setting the req->udata.buf pointer to point to an OTData structure, and setting the req->udata.len value to kNetbufDataIsOTData

The `sequence` field should be set to a non-zero value that will uniquely identify this request from all other oustanding requests on the endpoint.

The only defined flag for SndRequest is the `T_MORE` flags. The `T_MORE` flag indicates that more request data will be sent in a subsequent `SndRequest` call.

If the endpoint is in non-blocking or asynchronous mode, the `SndRequest` function will return a kOTFlowErr if flow control restrictions prevent the data from being accepted by the transport provider at the time the function is issued. After this error occurs, a T_GODATA event will be issued when the flow control restrictions are lifted. This error will never be returned if the endpoint is in blocking mode.

The behavior of SndRequest is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | kOTFlowErr never returned<br>Returns when flow control lifts and the reply has been received. |
| **Sync/Non-Blocking** | kOTFlowErr may be returned<br>Returns if flow control restrictions are in effect or when the reply has been received. |
| **Async/Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately<br>T_REPLY events may be sent to the notification routine as reply data becomes available |
| **Async/Non-Blocking** | kOTFlowErr may be returned<br>Returns to caller immediately<br>T_REPLY events may be sent to the notification routine as reply data becomes available |

**VALID STATES**

T_DATAXFER, T_INREL

**RESULT CODES**

kOTBadFlagErr
kOTBufferOverflowErr
kOTCanceledErr
kOTFlowErr
kOTLookErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

RcvRequest, SndReply, RcvReply






# RcvRequest

**FUNCTION**

RcvRequest — Read an incoming request.

**C INTERFACE**

```
OSStatus    OTRcvRequest(EndpointRef ref, TRequest* req, OTFlags*
            flags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvRequest(TRequest* req, OTFlags* flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| req->data.maxlen | x | / |
| req->data.len | / | x |
| req->data.buf | (?) | (x) |
| req->opt.maxlen | x | / |
| req->opt.len | / | x |
| req->opt.buf | ? | (?) |
| req->sequence | / | x |
| flags | / | x |

RcvRequest is used by a client of a connection-oriented transaction endpoint to read an incoming request. The client supplies a TRequest structure that will be filled out with the request data, options, and a sequence number to be passed back to the endpoint when a client makes a reply.

If the endpoint is in synchronous mode and is blocking, this function will wait for a request to arrive. If the endpoint is in asynchronous mode or is not blocking, the function will return any unread requests and the kOTNoDataErr result if there are no unread requests.

The endpoint will generate a T_REQUEST event when a request arrives. The client may poll for the arrival of a request by making the Look function or repeatedly calling this function for as long as the kOTNoDataErr result is returned. If the client has a notification routine installed on the endpoint, the event will be sent to the notification routine .

The TRequest structure has the following members:

```
struct TNetbuf data;
struct TNetbuf opt;
OTSequence sequence;
```

The data member will be filled in with the request data. The data.buf field points to a client-supplied buffer of size data.maxlen to hold the request.

The sequence member will be filled out with the endpoint-assigned sequence number for the transaction. This sequence number must be passed by the client to the SndReply function when sending the transaction reply.

The flags parameter will be filled in with the T_MORE flag if the client-supplied buffer is not large enough to hold the entire request. The client must reissue the RcvRequest function to retrieve the rest of the request. The options field will be ignored on these calls to RcvRequest.

In addition, the flags parameter may also contain the T_PARTIALDATA bit. In this case, the request data being received is only partial, and there is more coming, but it has not yet arrived. The difference between T_MORE and T_PARTIALDATA is that the T_MORE indicates that there is more data, and the next call to RcvRequest will read that data, while the T_PARTIALDATA flag does not have that guarantee. The T_PARTIALDATA bit will only be set on the first call to RcvRequest. All subsequent calls to RcvRequest are guaranteed to continue reading the partial data in question until the RcvRequest returns without the T_MORE flag set.

**VALID STATES**

T_DATAXFER, T_OUTREL

**RESULT CODES**

kOTBufferOverflowErr
kOTCanceledErr
kOTLookErr
kOTNoDataErr
kOTNotSupportedErr
kOTOutStateErr

**SEE ALSO**

SndRequest






# SndReply

**FUNCTION**

SndReply — Send a reply.

**C INTERFACE**

```
OSStatus    OTSndReply(EndpointRef ref, TReply* reply, OTFlags flags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::SndReply(TReply* reply, OTFlags flags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| reply->data.maxlen | / | / |
| reply->data.len | x | / |
| reply->data.buf | (x) | / |
| reply->opt.maxlen | / | / |
| reply->opt.len | x | / |
| reply->opt.buf | (x) | / |
| reply->sequence | x | / |
| flags | x | / |

SndReply is used by the client of a connection-oriented transaction endpoint to send a reply in response to an incoming request. The client passes a TReply structure containing the following members:

```
struct TNetbuf data;
struct TNetbuf opt;
OTSequence sequence;
```

The data TNetbuf contains the reply data. The reply data is pointed to by the udata.buf field.

A client may send non-contiguous data by setting the reply->data.buf pointer to point to an OTData structure, and setting the reply->data.len value to kNetbufDataIsOTData

The opt TNetbuf contains any options that apply to the reply.

The client must pass the sequence number returned by the RcvRequest function in the sequence parameter.

The behavior of SndReply is summarized in the table below.

| | |
|---|---|
| **Sync/Blocking** | `kOTFlowErr` never returned<br>Returns when flow control lifts and the reply has been successfully sent or timed out. |
| **Sync/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns if flow control restrictions are in effect or when the reply has been successfully sent or timed out. |
| **Async/Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately<br>A `T_REPLYCOMPLETE` event is sent to the notification routine when the reply is successfully sent or timed out. |
| **Async/Non-Blocking** | `kOTFlowErr` may be returned<br>Returns to caller immediately<br>A `T_REPLYCOMPLETE` event is sent to the notification routine when the reply is successfully sent or timed out. |

If the reply was not successfully sent (i.e. timed out - only for acknowledged requests), the `result` parameter will be set to kETIMEDOUTErr. For unacknowledged requests, a T_REPLYCOMPLETE event will still be generated for asynchronous clients so that the logic is the same for replying to both acknowledged and unacknowledged requests.

The `cookie` parameter passed to the notification routine to indicate completion will be set to the `reply` parameter of the original request (In cases where T_MORE is used to send the reply in multiple "chunks", the first `TReply*` is used).

**VALID STATES**

T_DATAXFER, T_OUTREL






**RESULT CODES**

`kOTBadFlagErr`
`kOTBadReferenceErr`
`kOTBadSequenceErr`
`kOTBadSyncErr`
`kOTCanceledErr`
`kOTLookErr`
`kOTNotSupportedErr`
`kOTOutStateErr`

**SEE ALSO**

None

# RcvReply

**FUNCTION**

RcvReply — Receive a reply.

**C INTERFACE**

```
OSStatus    OTRcvReply(EndpointRef ref, TReply* reply, OTFlags*
            replyFlags);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::RcvReply(TReply* reply, OTFlags* replyFlags);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref (C only) | x | / |
| reply->opt.maxlen | x | / |
| reply->opt.len | / | x |
| reply->opt.buf | x | (x) |
| reply->udata.maxlen | x | / |
| reply->udata.len | / | x |
| reply->udata.buf | x | (x) |
| reply->sequence | x | x |
| replyFlags | x | (x) |

RcvReply is used by the client of a connection-oriented transaction endpoint to receive a reply generated by a call to SndRequest. The client passes a TReply structure containing the following members:

```
struct TNetbuf udata;
struct TNetbuf opt
OTSequence sequence;
```

The udata TNetbuf contains the buffer for the reply data. The reply data will be pointed to by the udata.buf field.

The opt TNetbuf contains any options that apply to the reply.

The sequence field will be filled in with the sequence number of the matching SndRequest.

The flags parameter will be filled in with the T_MORE bit set if the RcvReply call did not read all of the reply data.

Since it is not possible to know ahead of time which request the incoming reply will match, the client must be prepared to receive a reply to any outstanding request. One way to deal with this is to first call RcvReply with a zero value in reply->udata.maxlen. This will return the sequence number and the option information, as well as setting the T_MORE flag (unless an error occurred - then there is no data to read). Once the matching request and the appropriate reply buffer have been found, a second RcvReply may be issued to read the actual reply data. On these second and subsequent reads, the reply->opt.len field will be set to 0. It is guaranteed that once a reply has been partially read and set the T_MORE flag, subsequent calls to RcvReply will read from that same reply until all the data has been read.






In addition, the flags parameter may also contain the T_PARTIALDATA bit. In this case, the reply data being received is only partial, and there is more coming, but it has not yet arrived. The difference between T_MORE and T_PARTIALDATA is that the T_MORE indicates that there is more data, and the next call to RcvReply will read that data, while the T_PARTIALDATA flag does not have that guarantee. The T_PARTIALDATA bit will only be set on the first call to RcvReply.

Clients in asynchronous mode will receive a T_REPLY event to indicate that incoming reply data is available. Once this occurs, RcvReply should be called repeatedly to read data until it returns a kOTNoDataErr. Otherwise, no future T_REPLY events will be received.

If a transaction has timed out awaiting reply data, the RcvReply function will return a kETIMEDOUTErr, and the reply->sequence field will indicate which request timed out.

**VALID STATES**

T_IDLE

**RESULT CODES**

kOTBadSequenceErr
kOTCanceledErr
kOTLookErr
kOTNoDataErr

**SEE ALSO**

SndURequest

# CancelRequest

**FUNCTION**

`CancelRequest` Cancel an outstanding SndRequest.

**C INTERFACE**

```
OSStatus    OTCancelRequest(EndpointRef ref, OTSequence seq);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::CancelRequest(OTSequence seq);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `seq` | x | . |

`CancelRequest` is used by the client of a connection-oriented transaction endpoint to cancel an outstanding outgoing request function (a call to `SndRequest`). Calling this function tells the protocol that this transaction is no longer of interest, and allows it to free up any memory associated with this transaction There is no acknowledgment from this function. If the sequence number indicated is not a valid sequence number, then nothing will be done.

It is the responsibility of the client to destroy any data structures associated with the canceled request.

If the value of the `seq` parameter is set to zero, then ALL outstanding SndRequest functions will be canceled.

Be sure to call `CancelReply` if you are canceling an incoming request. This is necessary because the `OTSequence` value of incoming requests are generated by the protocol, while the `OTSequence` value of outgoing requests are generated by the client, and they may overlap.

**VALID STATES**

T_DATAXFER

**RESULT CODES**

`kENOSRErr`

**SEE ALSO**

`CancelReply`






# CancelReply

**FUNCTION**

`CancelReply` Cancel an outstanding RcvRequest.

**C INTERFACE**

```
OSStatus    OTCancelReply(EndpointRef ref, OTSequence seq);
```

**C++ INTERFACE**

```
OSStatus    TEndpoint::CancelReply(OTSequence seq);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `seq` | x | . |

`CancelReply` is used by the client of a connection-oriented transaction endpoint to cancel an outstanding incoming request (received by RcvRequest). Calling this function tells the protocol that this transaction is no longer of interest, and allows it to free up any memory associated with this transaction. There is no acknowledgment from this function. If the sequence number indicated is not a valid sequence number, then nothing will be done.

It is the responsibility of the client to destroy any data structures associated with the canceled request.

If the value of the `seq` parameter is set to zero, then ALL outstanding incoming requests will be canceled.

Be sure to call `CancelRequest` if you are canceling an outgoing request. This is necessary because the `OTSequence` value of incoming requests are generated by the protocol, while the `OTSequence` value of outgoing requests are generated by the client, and they may overlap.

**VALID STATES**

T_DATAXFER

**RESULT CODES**

`kENOSRErr`

**SEE ALSO**

`CancelRequest`

# Address Mapping

Another type of Open Transport provider described in this document is called a mapper. A mapper is used to handle mapping names to protocol addresses. A client may create multiple mapper objects if desired.

With a mapper, a client can browse the network using the `LookupName` function to browse for all protocol addresses associated with a particular name or name pattern. A client can register a name on the network and make it visible to other network devices by using the `RegisterName` function. This registered name can be removed using the mapper's `RemoveName` function. The `ConfirmName` function is used to efficiently verify that a particular name to protocol address mapping is valid.

Not all protocol families can support all of the functions available to the mapper, but most will support the `LookupName` and `ConfirmName` functions (described below).

Mappers support the same functionality supplied to all providers.

As with all Open Transport providers, there are functions to create the mapper, `OpenMapper` and `AsyncOpenMapper`, and a function to destroy the mapper, `CloseProvider`. A mapper can be used in either synchronous or asynchronous mode and uses a notification routine to handle client callbacks for completion events. Unlike an endpoint, a mapper does not have any notification events that it sends to the client's notification routine. It has only completion events.

There are no functions available to cancel outstanding asynchronous mapper functions. The only way to cancel outstanding asynchronous mapper functions is to close the mapper (call `OTCloseProvider` with the MapperRef value).






# OTOpenMapper

**FUNCTION**

`OpenMapper` Create a mapper.

**C INTERFACE**

```
MapperRef   OTOpenMapper(OTConfiguration* config, OTOpenFlags oflag,
            OSStatus* err)
```

**C++ INTERFACE**

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

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `config` | x | / |
| `oflag` | x | / |
| `err` | / | x |

`OTOpenMapper` creates a mapper based on the supplied information, and returns a value by which the created mapper can be identified when calling other mapper functions.

The mapper will be opened in synchronous, non-blocking mode.

The `config` parameter is a pointer to an OTConfiguration structure. The client should not create one of these structures, but rather use the function OTCreateConfiguration:

```
pascal OTConfiguration* OTCreateConfiguration(char* path);
```

This function takes a string parameter which describes the desired provider layering (see the section on provider layering) and creates an `OTConfiguration` structure, returning a pointer to it to the client. The client should pass this pointer to `OpenMapper`. The `OTOpenMapper` function will destroy the structure. An example of calling `OpenMapper` using this function is shown below:

```
OSStatus err;
MapperRef ep = OTOpenMapper(OTCreateConfiguration("nbp"), 0, &err);
```

The name string passed to the OTCreateConfiguration function is dependent upon which protocol family the client wishes to create a mapper for. The name string to be used to create a mapper for a particular protocol family will be given in the Open Transport documentation for that particular protocol family.

The parameter `oflag` is not currently used and should be set to zero.

The output parameter `err` points to a result code.

**RESULT CODES**

`kENOENTErr`

`kENXIOErr`

`kENOMEMErr`

**SEE ALSO**

AsyncOpenMapper, CloseProvider






# OTAsyncOpenMapper

**FUNCTION**

AsyncOpenMapper Create a mapper.

**C INTERFACE**

```
OSStatus    OTAsyncOpenMapper(OTConfiguration* config, OTOpenFlags
            oflag, OTNotifyProcPtr proc, void* contextPtr)
```

**C++ INTERFACE**

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

**DESCRIPTION**

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

OTAsyncOpenMapper creates a mapper asynchronously, based on the supplied information. If this function returns an error immediately, then the notification function will not be called. If kOTNoError is returned, then the notification function will be called with the results of the open.

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

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

The mapper will be opened in asynchronous, non-blocking mode, and will already have a notification routine installed, which is the same notification routine used for the open. If you want a different notifier installed, use RemoveNotifier to remove the current one, and use InstallNotifier to install a new one.

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

**RESULT CODES**

kENOENTErr

kENXIOErr

kENOMEMErr

**SEE ALSO**

OTOpenMapper, OTCloseProvider

# RegisterName

**FUNCTION**

`RegisterName` Register a name on the network.

**C INTERFACE**

```
OSStatus    OTRegisterName(MapperRef ref, TRegisterRequest* request,
            TRegisterReply* reply);
```

**C++ INTERFACE**

```
OSStatus    TMapper::RegisterName(TRegisterRequest* req, TRegisterReply*
            reply);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->name.maxlen` | / | / |
| `req->name.len` | x | / |
| `req->name.buf` | (x) | / |
| `req->addr.maxlen` | / | / |
| `req->addr.len` | x | / |
| `req->addr.buf` | (x) | / |
| `req->flags` | x | / |
| `reply->addr.maxlen` | x | / |
| `reply->addr.len` | / | x |
| `reply->addr.buf` | / | (x) |
| `reply->nameid` | / | x |

`RegisterName` makes a name visible on the network to other network devices. Not all protocol families support dynamic name registration (such as TCP/IP), although this function may be still be implemented as a local function.

Note that most protocol implementations under Open Transport allow a client to specify a name in the call to the endpoint function `Bind`. This also causes the protocol to register a name on the network. This is a simpler technique and is preferable over having the client create and use a mapper object, especially where the name is associated with a server or service related to the created endpoint.

The `req->name` field is a `TNetbuf` that references the network name that is to be registered. The `req->addr` field is a `TNetbuf` structure that references the protocol address with which the name should be associated. The `req->addr` parameter may have a length of 0, and the underlying protocol will perform some default action. The format of the names and protocol addresses are specific to the underlying protocol, and their formats are described in the Open Transport documentation for that particular protocol family.

The `req->flags` field is used to control registration, where appropriate. Normally, this field is set to 0 for default registration behavior. For some protocols (e.g. the Netware naming service), a value may be set in this field. See the documentation for the naming service you are using to determine how to use this field.






If the mapper is in asynchronous mode, `RegisterName` returns immediately. Later, when the function completes execution, a `T_REGNAMECOMPLETE` event is issued. The `cookie` parameter to the notification routine has a value equal to that of the `name` parameter passed to the `RegisterName` call.

If the `reply` parameter is non-NULL, when the registration completes, the `reply->nameid` field will be set to a unique identifier for the registered name. This identifier can later be used to delete the name. This technique is more convenient than saving the name away somewhere so that it can be deleted later. If the `reply->addr.maxlen` field is large enough, the address actually registered will be returned. If this field is set to 0, then the address will not be filled in, and a `kOTNoError` result will be returned. If this field is not set to 0, then a `kOTBufferOverflowErr` will be returned if it is not large enough.

If the name was already registered, a `kOTAddressBusyErr` error is returned, and no information will be returned in the reply parameter.

**RESULT CODES**

`kOTBadNameErr`

`kOTAddrBusyErr`

`kOTNotSupportedErr`

`kOTBufferOverflowErr`

**SEE ALSO**

LookupName, ConfirmName, DeleteName

# DeleteName

**FUNCTION**

`DeleteName` Remove a previously registered name

**C INTERFACE**

```
OSStatus    OTDeleteName(MapperRef ref, TNetbuf* name);
```

**C++ INTERFACE**

```
OSStatus    TMapper::DeleteName(TNetbuf* name);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `name->maxlen` | / | / |
| `name->addr.len` | x | / |
| `name->buf` | ? | (?) |

`DeleteName` removes a name that was previously register. Not all protocol families support dynamic name registration and deletion (such as TCP/IP), although this function may be still be implemented as a local function.

The name parameter is a pointer to a TNetbuf that references the network name that is to be removed. The format of the names and protocol addresses are specific to the underlying protocol, and their formats are described in the Open Transport documentation for that particular protocol family.

If the mapper is in asynchronous mode, `DeleteName` returns immediately. Later, when the function completes execution, a `T_DELNAMECOMPLETE` event is issued. The `cookie` parameter to the notification routine has a value equal to that of the `name` parameter passed to the `DeleteName` call.

If the name was not found, a `kOTNoAddressErr` is returned.

**RESULT CODES**

`kOTBadNameErr`

`kOTCanceledErr`

`kOTNoAddressErr`

`kOTNotSupportedErr`

**SEE ALSO**

RegisterName






# DeleteName

**FUNCTION**

`DeleteName` Remove a previously registered name

**C INTERFACE**

```
OSStatus    OTDeleteNameByID(MapperRef ref, OTNameID id);
```

**C++ INTERFACE**

```
OSStatus    TMapper::DeleteName(OTNameID id);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `id` | x | / |

`DeleteName` removes a name that was previously register. Not all protocol families support dynamic name registration and deletion (such as TCP/IP), although this function may be still be implemented as a local function.

The `id` parameter is the `OTNameID` value that was returned when the name was registered.

If the mapper is in asynchronous mode, `DeleteName` returns immediately. Later, when the function completes execution, a `T_DELNAMECOMPLETE` event is issued. The `cookie` parameter to the notification routine has a value equal to that of the `id` parameter passed to the `DeleteName` call.

If the name was not found, a `kOTNoAddressErr` is returned.

**RESULT CODES**

`kOTBadNameErr`

`kOTCanceledErr`

`kOTNoAddressErr`

`kOTNotSupportedErr`

**SEE ALSO**

RegisterName

# LookupName

**FUNCTION**

`LookupName` Lookup a name or name pattern on the network

**C INTERFACE**

```
OSStatus    OTLookupName(MapperRef ref, TLookupRequest* req,
            TLookupReply* reply);
```

**C++ INTERFACE**

```
OSStatus    TMapper::LookupName(TLookupRequest* req, TLookupReply*
            reply);
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `ref` (C only) | x | / |
| `req->name.maxlen` | / | / |
| `req->name.len` | x | / |
| `req->name.buf` | (x) | / |
| `req->addr.maxlen` | / | / |
| `req->addr.len` | x | / |
| `req->addr.buf` | (x) | / |
| `req->maxcnt` | x | / |
| `req->timeout` | x | / |
| `reply->names.maxlen` | x | / |
| `reply->names.len` | / | x |
| `reply->names.buf` | (?) | (x) |
| `reply->rspcount` | / | x |

`LookupName` is used to find all protocol addresses that correspond to a particular name or name pattern.

The `req->maxcnt` parameter gives the maximum number of names that should be returned. If a client is expecting a specific number of replies for a particular name being looked up (usually one), then the client can get faster execution of the name lookup by specifying the expected number of replies.

The `req->timeout` parameter gives the approximate amount of time, in milliseconds that the lookup should attempt to find the requested number of names.

The `req` parameter is a pointer to a structure that supplies the information for the lookup request. The `req->name` TNetbuf supplies the network name that is to be looked up. Some protocol families support 'wild-cards' or pattern matching in this name.

The `req->addr` TNetbuf references the protocol address where the name(s) is(are) expected. This parameter is normally supplied with a length of 0, and the underlying protocol will search the default locations (where default depends upon the protocol family). However, if a client wants to look for a name(s) on a particular device, its protocol address can be specified. This MAY provide less network traffic.






The client passes a pointer to `TLookupReply` structure in the `reply` parameter containing the following members:

```
struct TNetbuf names;
unsigned long rspcount;
```

The `udata` `TNetbuf` should be initialized referencing a buffer area large enough to hold all of the expected replies (although see the note below if using the mapper in asynchronous mode.) Upon completion of the function, the `len` field of the udata `TNetbuf` will hold the total length of all the names found, and the `rspcount` field will contain the number of names that were found. Because there may be multiple names of varying lengths returned, each name returned is in the following format:

```
unsigned short    addrLen;      /* length of address which follows          */
unsigned short    nameLen;      /* length of name which follows             */
unsigned char     addr[];       /* address */
unsigned char     name[];       /* name, with pad-byte to quad boundary     */
```

Additionally, each name is aligned in the buffer so that the total length of the name (including 4 bytes for the two length fields) is a multiple of four bytes long. A client that is looking through the replies must account for this padding. For example,

```
len = ((short*)ptr)[0] + ((short*)ptr)[1];
ptr += (len + 3) & ~3;
```

The format of the names and protocol addresses are specific to the underlying protocol, and their formats are described in the Open Transport documentation for that particular protocol family.

If the mapper is in asynchronous mode, `LookupName` returns . Later, when the function completes execution, a `T_LKUPNAMECOMPLETE` event is issued. The `cookie` parameter to the notification routine has a value equal to that of the `reply` parameter passed to the `LookupName` call.

**Note:** In asynchronous mode, this function operates slightly different than other endpoint and mapper functions. In addition to the standard completion event, there is a `T_LKUPNAMERESULT` issued every time an additional name is added to the clients reply buffer. The client may choose to ignore this event, or the client may copy the reply from the reply buffer and set the `reply->udata.len` field or the `reply->rspcount` back to zero (but this may only be done inside the notification routine or results will be unpredictable). Using this feature, a client can avoid having to create a buffer large enough to hold all of the replies and reference it in the `reply->udata` `TNetbuf`. When dispatching the last name, the client may receive both a `T_LKUPNAMERESULT` and a `T_LKUPNAMECOMPLETE`, or may just receive a `T_LKUPNAMECOMPLETE`. If either the `reply->udata.len` or the `reply->rspcount` field is set back to zero, Open Transport will automatically set the other field to zero as well.

**RESULT CODES**

`kOTBadNameErr`

`kOTNoDataErr`

`kOTBufferOverflowErr`

`kOTNotSupportedErr`

**SEE ALSO**

DeleteName, ConfirmName

# Utility functions

This section describes the utility functions available to clients of Open Transport. Several of these functions have already been referenced, like `OTCreateConfiguration`.

## InitOpenTransport

**FUNCTION**

`InitOpenTransport` Initialize the Open Transport library

**C INTERFACE**

```
OSStatus    InitOpenTransport()
```

**DESCRIPTION**

`InitOpenTransport` must be called before your application or code resource can make Open Transport calls (see the Getting Started section for details). Make sure that you link with the appropriate library for your usage - applications must link with a different Open Transport library than code resource. (In case you're wondering, this is so that Open Transport can determine where to get memory on your behalf, and how to watch for your death).

This function returns an error code if Open Transport could not be initialized.

**SEE ALSO**

`CloseOpenTransport`






## CloseOpenTransport

**FUNCTION**

`CloseOpenTransport` Inform Open Transport that it will no longer be used.

**C INTERFACE**

```
void        CloseOpenTransport()
```

**DESCRIPTION**

`CloseOpenTransport` is called when the application or code resource will no longer use Open Transport functions. It is not necessary for applications to make this call, but it is **required** that stand-alone code resources do before they unload from memory.

An application can call `CloseOpenTransport` at any time. This potentially allows Open Transport to unload from memory until it is needed again. A subsequent call to `InitOpenTransport` will reload Open Transport.

**SEE ALSO**

`InitOpenTransport`

## OTCreateConfiguration

**FUNCTION**

`OTCreateConfiguration` Create an OTConfiguration structure for opening a provider

**C INTERFACE**

```
OTConfiguration*  OTCreateConfiguration(const char* path)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `path` | x | / |

`OTCreateConfiguration` takes a string which describes the provider desired (see the section on Specifying Provider layering) and returns a pointer to an `OTConfiguration` structure that can be used to open a provider.

Because this function is often called inline with the open call , it does not return an error. Instead, it returns a NULL if there is not enough memory to create the `OTConfiguration` structure, and it returns an ((`OTConfiguration`*)-1L) if the path that was passed to it was not parseable. The open routines check for these values, and return the appropriate error code.

**SEE ALSO**

`OTCloneConfiguration, OTDestroyConfiguration`

# OTCloneConfiguration

**FUNCTION**

`OTCloneConfiguration` Create a copy of an OTConfiguration structure

**C INTERFACE**

```
OTConfiguration*  OTCloneConfiguration(OTConfiguration* cfig)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| cfig | x | / |

`OTCloneConfiguration` returns a copy of the `OTConfiguration` passed in as a parameter. If the OTConfiguration* supplied is NULL or -1L, the same value will be returned.

**SEE ALSO**

`OTCreateConfiguration, OTDestroyConfiguration`






# OTDestroyConfiguration

**FUNCTION**

`OTDestroyConfiguration` Destroy an OTConfiguration structure

**C INTERFACE**

```
void        OTDestroyConfiguration(OTConfiguration* cfig)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| cfig | x | / |

`OTDestroyConfiguration` destroys the `OTConfiguration` passed in as a parameter and release all memory it has allocated.

It is rarely necessary to use this function because using an `OTConfiguration` to open a provider destroys the `OTConfiguration`. However, there may be cases where an `OTConfiguration` is created and never used. In this case, `OTDestroyConfiguration` should be called to free up the memory associated with the `OTConfiguration`.

**SEE ALSO**

`OTCloneConfiguration, OTCreateConfiguration`

# OTCreateOptions

**FUNCTION**

`OTCreateOptions` Create a `TNetbuf` for `OptionManagement` calls from a string specifying the options and values.

**C INTERFACE**

```
OSStatus    OTCreateOptions(const char* endptName, char** strPtr,
            TNetbuf* buf)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| endptName | x | x |
| strPtr | (x) | (?) |
| buf->len | x | x |
| buf->maxlen | x | x |
| buf->buf | x | (x) |

`OTCreateOptions` creates a `TNetbuf` suitable for use by calls that specify `options`.

The `endptName` parameter specifies the name of the endpoint for which the options are destined.

The `strPtr` parameter points to a pointer containing the option string information. If an error occurs, `strPtr` will be updated to point to the position in the string where the error occurred.

The `buf` parameter must point to a `TNetbuf` which has enough room to hold the requested options. Typically, the `buf->len` value is set to zero. If it is not, the option information will be appended to the TNetbuf beginning at the offset specified by `buf->len`. The `buf->len` value will be updated to reflect the new length when the function returns.

Not all endpoints support this functionality. If not, a `kOTNotSupportedErr` will be returned.

```
char* str = "BaudRate = 9650 DataBits = 8 Parity = 0 StopBits = 10";
UInt8 buffer[512];
TOptMgmt cmd;
cmd.opt.len = 0;
cmd.opt.maxlen = sizeof(buffer);
cmd.opt.buf = buffer;
cmd.flags = T_NEGOTIATE;
err = OTCreateOptions("serialA", &str, &cmd.opt)
```

The names for options are described in the documents for the specific providers. Value for options come in 3 basic flavors: numeric, string, and byte arrays.

The format for numeric options is:

an optional '-' for negative numbers

a leading '$' or "0x" for hexadecimal numbers,

followed by the digits comprising the number






The format for string options is:

a delimiter character - the first non-blank character after the "=" is the delimiter.

the characters comprising the string

the delimiter character repeated.

The format for byte array options is:

a leading "$" or "0x"

a sequence of hex digits with no intervening spaces or tabs. There must be an even number of digits.

**RESULT CODES**

`kOTNotSupportedErr`

`kOTBufferOverflowErr`

`kOTBadOptionErr`

**SEE ALSO**

`OTCreateOptionString`, `OptionManagement`

# OTCreateOptionString

**FUNCTION**

`OTCreateOptionString` Create a string from a TOption structure.

**C INTERFACE**

```
OSStatus    OTCreateOptionString(const char* endptName, TOption**
            optPtr, void* bufEnd, char* string, size_t stringSize)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| endptName | x | / |
| optPtr | (x) | (x) |
| bufEnd | x | / |
| string | x | (x) |
| stringSize | x | / |

`OTCreateOptionString` attempts to return a string corresponding to the option or options stored at `*optPtr`.

The `endptName` parameter specifies the name of the endpoint that the options are for.

The bufEnd parameter is a pointer to the byte of memory past the last option.

The `string` and `stringSize` parameters describe the character buffer where the string will be stored.

```
TOptMgmt   cmd;
UInt8      buffer[512];
char       string[256];
//
// Read the current settings
//
cmd.opt.len = sizeof(TOption);
cmd.opt.maxlen = sizeof(buffer);
cmd.opt.buf = buffer;
((TOption*)buffer)->len = sizeof(TOption);
((TOption*)buffer)->level = COM_SERIAL;
((TOption*)buffer)->name = T_ALLOPT;
((TOption*)buffer)->status = 0;
cmd.flags = T_CURRENT;
OptionManagement(theEndpt, &cmd, &cmd);
//
// Convert the returned options back into a string
//
TOption* opts = (TOption*)cmd.opt.buf;
err = OTCreateOptionString("serialA", &opts, cmd.opt.buf + cmd.opt.len,
                           string, sizeof(string));
printf("Options = \"%s\"", string);
```






**RESULT CODES**

`kOTNotSupportedErr`
`kOTBufferOverflowErr`
`kOTBadOptionErr`

**SEE ALSO**

`OTCreateConfiguration, OTDestroyConfiguration`

# OTEnterInterrupt

**FUNCTION**

`OTEnterInterrupt` Inform Open Transport that the caller is inside an interrupt routine

**C INTERFACE**

```
void        OTEnterInterrupt()
```

**DESCRIPTION**

`OTEnterInterrupt` informs Open Transport that it is at primary interrupt time. This allows Open Transport to more intelligently schedule network activity. A client must make this call whenever it is going to call Open Transport routines from primary interrupt time (e.g. from a VBL or Time Manager task, etc.)

`OTLeaveInterrupt` must be called before the client returns from the interrupt routine.

**NOTE**: On 68K Macintoshes, your A5 world must be set correctly before making this call (i.e. the same value it had when `InitOpenTransport` was called).

```
{
    OTEnterInterrupt();
    DoMyThing(myParms)
    OTLeaveInterrupt();
}
```

**SEE ALSO**

`OTLeaveInterrupt`






# OTLeaveInterrupt

**FUNCTION**

`OTLeaveInterrupt` Inform Open Transport that the caller is leaving an interrupt routine

**C INTERFACE**

```
void        OTLeaveInterrupt()
```

**DESCRIPTION**

`OTLeaveInterrupt` informs Open Transport that the caller will no longer be using Open Transport from within an interrupt routine (see `OTEnterInterrupt` on the previous page). A client should never make the `OTEnterInterrupt` call without the matching `OTLeaveInterrupt` call.

**NOTE**: On 68K Macintoshes, your A5 world must be set correctly before making this call (i.e. the same value it had when `InitOpenTransport` was called).

**SEE ALSO**

`OTEnterInterrupt`

# OTIdle

**FUNCTION**

`OTIdle` A routine to call when there is nothing else to do

**C INTERFACE**

```
void        OTIdle()
```

**DESCRIPTION**

`OTIdle` is a function that a client can call while it is waiting for asynchronous provider operations to complete. It is not necessary for the correct operation of Open Transport to call this function.

`OTIdle` may not be called at primary interrupt time. OTIdle will NOT call SystemTask , WaitNextEvent, or GetNextEvent.

**SEE ALSO**

`OTDelay`






# OTDelay

**FUNCTION**

`OTDelay` Delay for a specified number of seconds

**C INTERFACE**

```
void        OTDelay(UInt32 seconds)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| seconds | x | / |

`OTDelay` will delay for the number of seconds specified in the seconds parameter. While the delay is occurring, it will continuously call `OTIdle`.

`OTDelay` should only be called from within an application at SystemTask time, and there is no known reason for calling it. It is provided for compatibility with the UNIX `sleep` function (`sleep` is #defined to be the `OTDelay` function).

**SEE ALSO**

`OTIdle`

# OTGetIndexedPort

**FUNCTION**

`OTGetIndexedPort` Return information on the n'th hardware port available to Open Transport

**C INTERFACE**

```
Boolean     OTGetIndexedPort(OTPortRecord* record, size_t index)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| record | x | (x) |
| index | x | / |

`OTGetIndexedPort` is used to iterate through the hardware ports available to Open Transport. Repeated calls to `OTGetIndexedPort with incrementing index numbers (starting with 0)` will return information on each of the available hardware ports in turn. The function will return false if the index value is outside the range of available ports.

The `record` parameter will be filled in with the port information if `OTGetIndexedPort` returns true.

See the section on ports for information on the `OTPortRecord` structure.

**SEE ALSO**






# OTFindPort

**FUNCTION**

`OTFindPort` Find information about a port given the name of the port

**C INTERFACE**

```
Boolean     OTFindPort(OTPortRecord* record, const char* portName)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| record | x | (x) |
| portName | (x) | / |

`OTFindPort` returns information into the `record` parameter about the port named in `portName`.

This function returns `false` if a port with the specified name does not exist.

```
if ( OTFindPort(recordBuf, "serialA") )
{
    // Do something with record information
}
```

**SEE ALSO**

`OTFindPortByRef`

# OTFindPortByRef

**FUNCTION**

`OTFindPortByRef` Find information about a port given the OTPortRef value.

**C INTERFACE**

```
Boolean     OTFindPortByRef(OTPortRecord* record, OTPortRef ref)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| record | x | (x) |
| ref | x | / |

`OTFindPortByRef` returns information into the `record` parameter about the port `OTPortRef` is `ref`.

The function returns false if a port matching the specifications does not exist.

```
OTPortRef ref = OTCreatePortRef(kOTMotherboardBus, kOTSerialDevice, 0, 10);
if ( OTFindPortByRef(recordBuf, ref,) )
{
    // Do something with record information
}
```

See the section on Ports for more information about `OTPortRefs` and device names.

**SEE ALSO**

`OTFindPort`






# OTCreatePortRef

**FUNCTION**

`OTCreatePortRef` Create an OTPortRef, given knowledge about the port.

**C INTERFACE**

```
OTPortRef   OTCreatePortRef(UInt8 busType, UInt16 devType, UInt16 slot,
            UInt16 other)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| busType | x | / |
| devType | x | / |
| slot | x | / |
| other | x | / |

`OTCreatePortRef` creates an `OTPortRef` that can be used for the various port-finding utilities of Open Transport. An `OTPortRef` contains the 4 pieces of information stored in an opaque 32-bit format. Manipulation of `OTPortRef` values should only be done with the utility functions provided by Open Transport.

Currently defined bus types are:

```
enum
{
      kOTUnknownBusPort = 0,
      kOTMotherboardBus = 1,
      kOTNuBus          = 2,
      kOTPCIBus         = 3,
      kOTGeoPort        = 4,
      kOTPCMCIABus      = 5
};
```

These definitions can be found in the OpenTransport.h header file.

Currently defined device types are:

```
enum
{
    kOTADEVDevice         = 1,          /* An Atalk ADEV            */
    kOTMDEVDevice         = 2,          /* A TCP/IP MDEV            */
    kOTLocalTalkDevice    = 3,          /* LocalTalk                */
    kOTIRTalkDevice       = 4,          /* IRTalk                   */
    kOTTokenRingDevice    = 5,          /* Token Ring               */
    kOTISDNDevice         = 6,          /* ISDN                     */
    kOTATMDevice          = 7,          /* ATM                      */
    kOTSMDSDevice         = 8,          /* SMDS                     */
    kOTSerialDevice       = 9,          /* Serial                   */
    kOTEthernetDevice     = 10,         /* Ethernet                 */
    kOTSLIPDevice         = 11,         /* SLIP Pseudo-device       */
    kOTPPPDevice          = 12,         /* PPP Pseudo-device        */
    kOTModemDevice        = 13,         /* Modem Pseudo-Device      */
    kOTFastEthernetDevice = 14,         /* 100 MB Ethernet          */
    kOTFDDIDevice         = 15,         /* FDDI                     */
    kOTATMLANEDevice      = 16,         /* ATM LAN emulation        */
    kOTATMSNAPDevice      = 17          /* ATM SNAP emulation       */
};
```

These definitions can be found in the OpenTptLinks.h header file.

The slot parameter typically defines the slot location of the device, while the other parameter typically disambiguates between multiple hardware ports within that slot, if necessary.

**SEE ALSO**

`OTGetDeviceTypeFromPortRef, OTGetBusTypeFromPortRef, OTGetSlotFromPortRef`






# OTGetDeviceTypeFromPortRef

**FUNCTION**

`OTGetDeviceTypeFromPortRef` Extracts the device type from an OTPortRef.

**C INTERFACE**

```
UInt16      OTGetDeviceTypeFromPortRef(OTPortRef ref)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref | x | / |

This function extracts the device type information from the OTPortRef and returns the value.

**SEE ALSO**

`OTGetBusTypeFromPortRef, OTGetSlotFromPortRef, OTCreatePortRef`

# OTGetBusTypeFromPortRef

**FUNCTION**

`OTGetBusTypeFromPortRef` Extracts the bus type from an OTPortRef.

**C INTERFACE**

```
UInt16      OTGetBusTypeFromPortRef(OTPortRef ref)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref | x | / |

This function extracts the bus type information from the OTPortRef and returns the value.

**SEE ALSO**

`OTGetDeviceTypeFromPortRef, OTGetSlotFromPortRef, OTCreatePortRef`






# OTGetSlotFromPortRef

**FUNCTION**

`OTGetSlotFromPortRef` Extracts the slot information from an OTPortRef.

**C INTERFACE**

```
UInt16      OTGetDeviceTypeFromPortRef(OTPortRef ref, UInt16* otherPtr)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| ref | x | / |
| otherPtr | x | (x) |

This function extracts the slot information from the OTPortRef. It returns the value of the slot number stored in the `OTPortRef`, and if `otherPtr` is not NULL, it store the "other" information at the value pointed to by `otherPtr`.

**SEE ALSO**

`OTGetBusTypeFromPortRef, OTGetDeviceTypeFromPortRef, OTCreatePortRef`

# OTCreateSystemTask

**FUNCTION**

`OTCreateSystemTask` Create a reference that will allow a function to be run at the next SystemTask.

**C INTERFACE**

```
long        OTCreateSystemTask(OTProcessProcPtr proc, void* contextInfo)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| proc | x | / |
| contextInfo | x | / |

`OTCreateSystemTask` creates a reference that can be used to schedule the function pointed to by `proc` to be called at the next SystemTask. When scheduled (see OTScheduleSystemTask), the function will be called back at the next SystemTask and will be passed the `contextInfo` as a parameter. For 68K clients only, at the time of the callback, the A5 global world will be set to the A5 global world at the time that `OTCreateSystemTask` was called. The typedef for the `OTProcessProcPtr` function is:

```
typedef pascal void (*OTProcessProcPtr)(void* contextInfo);
```

The return value of the function is a reference that should be used when calling `OTScheduleSystemTask` or `OTDestroySystemTask`. If the return value is zero, then there is not enough memory to allocate the necessary data.

**SEE ALSO**

`OTDestroySystemTask, OTScheduleSystemTask, OTCancelSystemTask`






# OTDestroySystemTask

**FUNCTION**

`OTDestroySystemTask` Destroy a SystemTask object created with the OTCreateSystemTask function

**C INTERFACE**

```
void        OTDestroySystemTask(long stCookie)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| stCookie | x | / |

`OTDestroySystemTask` makes a reference returned by `OTCreateSystemTask` invalid, and frees any resources associated with the `OTCreateSystemTask` call. This call should be made when it is no longer necessary to schedule the function. It can be made at any time.

**SEE ALSO**

`OTCreateSystemTask, OTScheduleSystemTask, OTCancelSystemTask`

# OTScheduleSystemTask

**FUNCTION**

`OTScheduleSystemTask` Schedule a function to be called at SystemTask time

**C INTERFACE**

```
Boolean      OTScheduleSystemTask(long stCookie)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `stCookie` | x | / |

`OTScheduleSystemTask` will schedule the function associated with the `stCookie parameter` (this value was returned by `OTCreateSystemTask`) for running at the next SystemTask time. This call can be made at any time. This function returns true if the function was schedule, false if not. If the function was not schedule, and the `stCookie` parameter is valid, then this indicates that the function is already scheduled to run.

**SEE ALSO**

`OTCreateSystemTask, OTDestroySystemTask, OTCancelSystemTask`






# OTCancelSystemTask

**FUNCTION**

`OTCancelSystemTask` Cancel a function that is scheduled to be called at SystemTask time

**C INTERFACE**

```
Boolean      OTCancelSystemTask(long stCookie)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `stCookie` | x | / |

`OTCancelSystemTask` will cancel a function that was scheduled to run at System Task time by calling `OTScheduleSystemTask`. The function returns true if the scheduling was able to be canceled. If the function returns false, then either the function was not scheduled or it is too late to cancel.

**SEE ALSO**

`OTCreateSystemTask, OTDestroySystemTask, OTScheduleSystemTask`

# OTCanMakeSyncCall

**FUNCTION**

| | |
|---|---|
| `OTCanMakeSyncCall` | Determine whether synchronous calls to Open Transport are allowed |

**C INTERFACE**

```
Boolean      OTCanMakeSyncCall()
```

**DESCRIPTION**

`OTCanMakeSyncCall` will return true if you can make a synchronous call to Open Transport. A false will be returned if a synchronous call will fail. This call will not function properly if you make the call from inside of an interrupt routine, and you haven't called `OTEnterInterrupt`.

**SEE ALSO**

`OTCreateSystemTask, OTDestroySystemTask, OTScheduleSystemTask`






# OTCreateDeferredTask

**FUNCTION**

| | |
|---|---|
| `OTCreateDeferredTask` | Create a reference that will allow a function to be run at the next Deferred task time. |

**C INTERFACE**

```
long        OTCreateDeferredTask(OTProcessProcPtr proc, void*
            contextInfo)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `proc` | x | / |
| `contextInfo` | x | / |

`OTCreateDeferredTask` creates a reference that can be used to schedule the function pointed to by `proc` to be called at the next Deferred Task time When scheduled (see `OTScheduleDeferredTask`), the function will be called back at the appropriate time and will be passed the `contextInfo` as a parameter. For 68K only, at the time of the callback, the A5 global world will be set to the A5 global world at the time that `OTCreateDeferredTask` was called. The typedef for the `OTProcessProcPtr` function is:

```
typedef void (*OTProcessProcPtr)(void* contextInfo);
```

The return value of the function is a reference that should be used when calling `OTScheduleDeferredTask` or `OTDestroyDeferredTask`. If the return value is zero, then there is not enough memory to allocate the necessary data.

**SEE ALSO**

`OTDestroyDeferredTask, OTScheduleSystemTask`

# OTDestroyDeferredTask

**FUNCTION**

`OTDestroyDeferredTask` Destroy a deferred task object created with the OTCreateDeferredTask function

**C INTERFACE**

```
void        OTDestroyDeferredTask(long dtCookie)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| dtCookie | x | / |

`OTDestroyDeferredTask` makes a reference returned by `OTCreateDeferredTask` invalid, and frees any resources associated with the `OTCreateDeferredTask` call. This call should be made when it is no longer necessary to schedule the function. It can be made at any time.

**SEE ALSO**

`OTCreateDeferredTask, OTScheduleDeferredTask`






# OTScheduleDeferredTask

**FUNCTION**

`OTScheduleDeferredTask` Schedule a function to be called at deferred task time

**C INTERFACE**

```
Boolean     OTScheduleDeferredTask(long dtCookie)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| dtCookie | x | / |

`OTScheduleDeferredTask` will schedule the function associated with the `dtCookie parameter` (this value was returned by `OTCreateDeferredTask`) for running at the next Deferred Task time. This call can be made at any time. This function returns true if the function was schedule, false if not. If the function was not schedule, and the `dtCookie` parameter is valid, then this indicates that the function is already scheduled to run.

This function is intended to be used by interrupt service routines to schedule deferred task processing independently of the underlying deferred task mechanism.

**SEE ALSO**

`OTCreateDeferredTask, OTDestroyDeferredTask`

# Native functions

This section describes some Open Transport functions that are only available to native clients. They have no mixed-mode glue associated with them, so your application or extension must be built FAT in order to use these APIs on a Power Macintosh machine.

## OTYieldPortRequest

**FUNCTION**

| | |
|---|---|
| `OTYieldPortRequest` | Ask all users of a port to yield the port |

**C INTERFACE**

```
OSStatus    OTYieldPortRequest(ProviderRef provider, OTPortRef ref,
            OTClientList* buffer, size_t bufferSize)
```

**DESCRIPTION**

| Parameters | Before Call | After Call |
|---|---|---|
| `provider` | x | / |
| `ref` | x | / |
| `buffer` | (?) | (x) |
| `bufferSize` | x | / |

`OTYieldPortRequest` is used to request the use of a port (normally, a serial port or modem) from whoever is currently using the port. If the function returns an error, then something went wrong and the request could not be completed. This could be because of lack of resources, or the port specified does not support the request. If `kEBUSYErr` is returned, the buffer parameter will contain a list of all of the clients that rejected the request. The bufferSize parameter is the size of the `buffer` parameter (including the `fNumClients` field). The buffer is a list of concatenated pascal strings enumerating the name of each client that rejected the request (normally only one).

```
struct OTClientList
{
       size_t fNumClients;
       UInt8  fBuffer[4];
};
```

In the case where you want to be extremely rude and grab the port anyway, pass NULL for the buffer pointer. In this case, if the function returns kOTNoError, then the port has been yielded, and you can grab it. Note that normally a port will only yield in this manner if it's current client is passively listening - the port will not be grabbed if a connection is in progress.

The supplied ProviderRef must be a provider (normally an endpoint) that is open on the requested port. You may not use this provider while the `OTYieldPortRequest` call is in progress.

This call may only be made at System Task time (OTCanMakeSyncCall returns true).






Once the yield port request returns kOTNoError (or kENOENTErr - see below), you may attempt to use the port (normally, this is by binding with a qlen <> 0 ,or by connecting. If you need to cancel the yield request, call `OTYieldPortRequest` with a buffer of (void*)-1L. If you do not do one of these three things, the port will automatically unyield in about 10 seconds.

**ERRORS**

| | |
|---|---|
| kOTBadSyncErr | Called at non-system-task time |
| kOTBadReferenceErr | Either the provider does not use the requested OTPortRef, or the requested OTPortRef does not exist. |
| kOTNotSupportedErr | The requested port does not support yielding. |
| kENOMEMErr | Not enough memory to complete the request |
| kEBUSYErr | A client "nak"d the request to yield |
| kENXIOErr | The underlying provider will not yield, probably because the current client is already connected. |
| kENOENTErr | The underlying provider does not currently have a client, so doing an OTYieldPortRequest was unnecessary. |

# Client callbacks

Open Transport provides several ways for clients to get notification when significant events occur in the Open Transport system. The first is through the notifiers that are installed on providers. Several events may be passed to your notifier that are not "standard" events. These are:

kOTProviderIsDisconnected

Your provider was a listener (bound with qlen <> 0), and it has been disconnected (is no longer listening). This currently only happens with serial ports, but could also happen with other connection-oriented drivers that have characteristics similar to serial ports. You will get a kOTProviderIsReconnected message when the cause of the disconnection is relieved (see the section on the OTYieldPortRequest function).

kOTProviderIsReconnected

Your provider has been reconnected. Your provider is once again listening.

kOTProviderWillClose

Your provider will be closed as soon as you return from the notifier. This callback is always done at System Task time (OTCanMakeSyncCall will return true). You may set your provider into synchronous mode and make a few last calls to prepare for being closed. Once you return from the notifier, any calls made using the provider except OTCloseProvider will fail with a kOTOutStateErr.

kOTProviderIsClosed

Your provider has been closed. The reason for being closed can be found in the OTResult value passed to your notifier. The reasons typically are kOTPortHasDiedErr, kOTPortWasEjectedErr, or kOTPortLostConnectionErr. Any calls other than OTCloseProvider will fail with a kOTOutStateErr.

The second type of callback that Open Transport supplies is general client notification. In order to receive these notifications, you must call the OTRegisterAsClient function:

```
OSStatus OTRegisterAsClient(OTClientName name, OTNotifyProcPtr proc)
```

This function supplies Open Transport with your name ("`name`" is currently a char* parameter - a zero-terminated "C" string. It should be read from a resource in order to allow the name to be internationalized). The `proc` parameter is a pointer to a notification procedure that will be called whenever significant events occur in the Open Transport system.

When you no longer want to receive notification, you can call `OTUnregisterAsClient`:

```
OSStatus OTUnregisterAsClient()
```

Calling this function is optional, since either calling CloseOpenTransport or just exiting your application will automatically unregister you as a client.






The events you may currently receive are:

kOTPortDisabled

A port has gone offline. The `OTResult` parameter will give the specific reason, if known, and the `cookie` parameter is the `OTPortRef` of the port that went offline. A port going offline also often results in providers getting `kOTProviderIsClosed` events. There is no guarantee in Open Transport as to which of these events will be received first.

kOTPortEnabled

A port which had previously been disabled is now enabled. The `cookie` parameter is the `OTPortRef` of the port that is now enabled.

kOTNewPortRegistered

A new port has been registered with Open transport. The `cookie` parameter is the `OTPortRef` of the new port.

kOTClosePortRequest

You currently are using a provider that is using a port that some other application wants to use. The OTResult parameter is the reason for the request (normally kOTNoError or kOTUserRequestedErr), and the cookie parameter is a pointer to an OTPortCloseStruct:

```
struct OTPortCloseStruct
{
        OTPortRef       fPortRef;
        ProviderRef     fTheProvider;
        OSStatus        fDenyReason;
        OTClientName    fRequestor;
};
```

The `fPortRef` field describes the port that is asking to be closed. The `fTheProvider` field tells you the provider that is currently using the port. The `fRequestor` field is the name of the requesting application or system service. If you will yield the port, you need do nothing. If you won't yield the port, you must fill in the fDenyReason field with a non-zero value that may specify the reason (normally, kOTUserRequestedErr is used). This callback is always done at SystemTask time, so you may put up a dialog to the user or take other action as appropriate. Currently, this callback is only used for serial ports, but it is applicable to any hardware device which cannot shared the port with multiple clients. If you are willing to yield the port, and you are currently actively connected (as opposed to listening in the T_IDLE state with a qlen <> 0), you must issue a synchronous `OTSndDisconnect` in order to yield the port.

Your provider will receive a `kOTPortIsDisconnected` event if the port is grabbed away from you. When the "grabber" is done, it will receive a `kOTPortIsReconnected` event.

# Advanced Topics

This section describes some topics for the more advanced clients of Open Transport.

## No-Copy Receives

Open Transport provides support for receives without copying the data. When using this feature of Open Transport, you should be aware that using no-copy receives can adversely affect the performance of the Open Transport system if it is not done correctly.

The `OTBuffer` data structure is the structure returned by no-copy receives. If you are familiar with STREAMS `mblk_t` data structures, you can see that this structure is just a slight modification of the `mblk_t` structure.

```
struct OTBuffer
{
        void*           fLink;          // b_next & b_prev
        void*           fLink2;
        OTBuffer*       fNext;          // b_cont
        UInt8*          fData;          // b_rptr
        size_t          fLen;           // b_wptr
        void*           fSave;          // b_datap
        UInt8           fBand;          // b_band
        UInt8           fType;          // b_pad1
        UInt8           fPad1;
        UInt8           fFlags;         // b_flag
};
```

| | |
|---|---|
| fLink | A link field, unused |
| fLink2 | Another link field, unused |
| fNext | A pointer to the next OTBuffer in the chain |
| fData | A pointer to the data belonging to this OTBuffer |
| fLen | The length of data pointed to by fData |
| fSave | A reserved field |
| fBand | A byte corresponding to the "band" of the data. |
| fType | A byte describing the "type" of the data (normally M_DATA, M_PROTO, or M_PCPROTO) |
| fPad1 | An unused byte |
| fFlags | The flags associated with the data (MSGMARK, MSGDELIM) |

By tracing the chain of fNext pointers, all of the data associated with the message can be accessed. Under NO CIRCUMSTANCES WRITE TO THIS DATA STRUCTURE. It is read-only. If you write to it, it is quite possible that you will crash the system. Under Copland, if you write to it, you will get an access fault which will kill your application.






Because this structure is read-only, Open Transport provides a few utilities to allow data to be read from this structure.

```
struct OTBufferInfo
{
        OTBuffer*       fBuffer;
        size_t          fOffset;
        UInt8           fPad;
};

typedef struct OTBufferInfo  OTBufferInfo;

#define OTInitBufferInfo(infoPtr, theBuffer)       \
        (infoPtr)->fBuffer    = theBuffer;         \
        (infoPtr)->fPad       = theBuffer->fPad1;  \
        (infoPtr)->fOffset    = 0
```

```
extern "C" Boolean OTReadBuffer(OTBufferInfo* info, void* buf, size_t* len);
extern "C" size_t OTBufferDataSize(OTBuffer* bfr)
extern "C" void OTReleaseBuffer(OTBuffer* bfr)
```

The `OTBufferInfo` structure keeps track of where you last left off in a buffer. This allows you to read pieces of the data into multiple buffers, keeping track of where you left off.

`OTReadBuffer` reads `*len` bytes from the `OTBuffer` description stored in `info` into the buffer `buf`. It returns true if the read request completely exhausted the bytes in the buffer, and it returns false if there are more bytes in the buffer to be read. In all cases, `*len` is updated with the actual number of bytes copied.

`OTBufferDataSize` returns the number of data bytes in the `OTBuffer` (including bytes in the `fNext` chain).

`OTReleaseBuffer` returns the buffer to the system when you are done with it.

Depending on the API being used, a no-copy receive is requested by using the constant `kOTNetbufDataIsOTBufferStar`.

```
enum
{
        kOTNetbufDataIsOTBufferStar  = (size_t)-3
};
```

A few examples follow:

```
{
        OTBuffer* myBuffer;
        OTResult result = Rcv(myEndpoint, &myBuffer, kOTNetbufDataIsOTBufferStar);
}
{
        OTBuffer*       myBuffer;
        TUnitData       data;
        OTFlags         flags;

        data.addr.buf           = &addrBuf;
        data.addr.maxlen        = sizeof(addrBuf);
        data.opt.maxlen         = 0;
        data.udata.buf          = (UInt8*)&myBuffer;

        data.udata.maxlen       = kOTNetbufDataIsOTBufferStar;

        OTStatus status = OTRcvUData(&data, &flags)
}
```

Once you have copied the data out of the OTBuffer, you should call OTReleaseBuffer to return it to Open Transport.

**WARNING**: In many cases, for performance reasons, drivers will pass up their actual DMA buffers. If this is the case, when you do a no-copy receive, you are getting the actual DMA buffers from the driver. If you hold on to the buffer for too long, you may begin to starve the driver for DMA buffers, which will adversely affect the performance of the system. It is very important that if you are doing a no-copy receive, you hold onto the buffer for as short a time as possible. If it is necessary to hold on to the buffer for any length of time, overall performance will be better if you make a copy of the data and return the buffer to the system.

# Autopush

Open Transport provides support for the autopush feature of STREAMS. Under SVR4.2, autopush information is base on device major numbers. On systems which allow dynamic loading of modules, this does not make sense. It would require that devices be loaded before autopush information could be configured. So, instead of using major numbers, Open Transport uses the device name, even though this does not match the SVR4.2 implementation.

Autopush is implemented by talking to the system administration STREAMS module, named "sad". All configuration is done by means of IOCTL command to the "sad" module. The following are the pertinent structures:






```
#define kSADModuleName       "sad"

enum
{
        kOTAutopushMax= 8,

        I_SAD_SAP       = MIOC_CMD(MIOC_SAD, 1),     /* Set autopush information  */
        I_SAD_GAP       = MIOC_CMD(MIOC_SAD, 2),     /* Get autopush information  */
        I_SAD_VML       = MIOC_CMD(MIOC_SAD, 3)      /* Validate a list of modules */
};

/* I_LIST structures */
struct str_list
{
        int                     sl_nmods;      /* number of modules in sl_modlist array */
        struct str_mlist*       sl_modlist;
};

struct str_mlist
{
        char    l_name[FMNAMESZ + 1];
};

struct OTAutopushInfo  /* Ioctl structure used for SAD_SAP and SAD_GAP commands */
{
        unsigned int   sap_cmd;
        char           sap_device_name[kMaxModuleNameSize];
        long           sap_minor;
        long           sap_lastminor;
        long           sap_npush;
        char           sap_list[kOTAutopushMax][kMaxModuleNameSize];
};

typedef struct OTAutopushInfo       OTAutopushInfo;

/*
 * Command values for sap_cmd
 */

enum
{
        kSAP_ONE              = 1,   /* Configure a single minor device         */
        kSAP_RANGE            = 2,   /* Configure a range of minor devices      */
        kSAP_ALL              = 3,   /* Configure all minor devices             */
        kSAP_CLEAR            = 4    /* Clear autopush information              */
};
```

Autopush information is set, retrieved and cleared by using sending IOCTLs to the "sad" driver. These IOCTLs may be either I_STR IOCTLs or "transparent" IOCTLs. In order to set or get autopush information, you must allocate an OTAutopushInfo structure.
To set autopush information about a module, set the `sap_cmd` field to one of the command values specificed above, set up the `sap_minor` and `sap_lastminor` numbers (if applicable) , and set the `sap_device_name` field to a "C"-style string containing the name of the device about which autopush information is desired. Fill in the `sap_npush` field with the number of modules to autopush, and then fill in the sap_list with the names of the modules to autopush. Then execute code that looks like the following:

```
struct strioctl      stri;
stri.ic_cmd = I_SAD_SAP;
stri.ic_dp     = &myOTAutopushInfoStructure;
stri.ic_len    = sizeof(OTAutopushInfo);
stri.ic_timout = -1;
err = OTStreamIoctl(myStreamRef, I_STR, (void*)&stri);

or

err = OTStreamIoctl(myStreamRef, I_SAD_SAP, (void*)&stri);
```

This will set the autopush information for the module. Use kSAP_ALL to configure all minor numbers of a device. This is normally the value that will be used to configure autopush. For specialty usage, you can use kSAP_ONE to configure a single minor number (specified in the sap_minor field) or kSAP_RANGE to configure a range of minor numbers (specified with the lower value in the sap_minor field and the higher value in the sap_lastminor field). The kSAP_CLEAR command can be used to clear a previous autopush. Set sap_minor to 0 to clear all autopush information for a module. Otherwise, you can clear a single range or minor number by specifying sap_minor and sap_lastminor value. The sap_lastminor field can be set to -1 to clear the range set that started with sap_minor. Note that you cannot clear ranges that overlap with ranges that were set by two individual kSAP_RANGE commands (you'll get a kEEXISTErr).

| | |
|---|---|
| kENODEVErr | the module specified is not configured for autopush. |
| kENOSTRErr | the module specified does not exist in the system |
| kEINVALErr | the command specified is not recognized by "sad" |
| kERANGEErr | the minor number range specified was invalid, or the range is not configured for autopush (when clearing a range). |
| kEEXISTErr | an attempt was made to configure or clear a minor number or minor number range that overlaps an existing range of minor numbers |

You can use the I_SAD_GAP command to obtain the autopush information about a module. Fill out the OTAutopushInfo structure for the module and minor number(s) desired, along with the appropriate command code (kSAP_ALL, kSAP_RANGE, or kSAP_ONE). and you will get back the OTAutopushInfo structure that was used to set that particular autopush. For instance, if you ask for a single minor number, and it was programmed as part of a range, you will get back the kSAP_RANGE or kSAP_ALL. information that was used to set the autopush information






# Ownership of Providers

Open Transport attempts to "death watch" providers (including endpoints, mappers, and services providers). If a client dies or quits without closing all it's outstanding providers, Open Transport attempts to "clean up" and close them on behalf of the client. This leads to an interesting problem. Every shared library, code resource, or application that creates an endpoint, or uses one of the endpoint functions that allocate memory on behalf of the client (list follows) must be a client of Open Transport (having called InitOpenTransport). For ASLM shared libraries and applications, Open Transport can deathwatch the library or application easily. For CFM shared libraries, the client MUST call CloseOpenTransport before terminating (this can be done by making CloseOpenTransport the termination procedure for the CFM library) Since Open Transport keeps track of the owner of all providers, and closes them when the owner terminates, some provision must be made for transfering the ownership of a provider. The function OTTransferProviderOwnership is intended for that purpose. You pass it a ProviderRef that you wish to obtain ownership of, and it will return a new ProviderRef that belongs to you. The old ProviderRef is then obsolete and should not be used:

```
pascal ProviderRef OTTransferProviderOwnership(ProviderRef oldRef, OTClient oldOwner,
OSStatus* errPtr);
```

You can use the OTWhoAmI function to obtain the OTClient value. In order to transfer the ownership of the endpoint, the prior owner will need to give you it's OTClient value through some API.

```
pascal OTClient OTWhoAmI(void);
```

# Appendix A - Sample Code

For an examples of Open Transport clients, see the disk “Open Transport Samples” in the Open Transport software distribution.

# Appendix B - Endpoint States

This appendix lists and describes the endpoint states that are visible to Open Transport clients. It also provides tables showing which Open Transport routines change the state of endpoints.

**Table B-1.** Open Transport Endpoint States

| State | Decimal value | Meaning |
|---|---|---|
| T_DATAXFER | 5 | This connection-oriented endpoint has a connection established; the endpoint can now send and receive data. |
| T_IDLE | 2 | The endpoint has been bound to a local protocol address. For non-connection-oriented endpoints, it is ready for use. For connection-oriented endpoints, the endpoint is ready to receive incoming connection requests, or for the client to initiate a connection. |
| T_INCON | 4 | The connection-oriented endpoint has received a connection request, and the client has not yet accepted (using the `Accept` function) or rejected (using the `SndDisconnect` function) the connection request. |
| T_INREL | 7 | The connection-oriented endpoint has received an incoming request for an orderly disconnect, and the client has not yet acknowledged the release (using the `RcvOrderlyDisconnect` function). The client may continue to send data on this endpoint, until acknowledging the release, but may no longer read incoming data. Not all endpoints support orderly disconnects. |
| T_OUTCON | 3 | The client has used the `Connect` function to initiate a connection request on a connection-oriented endpoint, and the connection has not yet been established. |
| T_OUTREL | 6 | The client has initiated an orderly disconnect (using the `SndOrderlyDisconnect` function), which the remote endpoint has not yet acknowledged. The client may continue to read data from the connection, but may not send any more data. Not all endpoints support orderly disconnects. |
| T_UNBND | 1 | The endpoint is initialized, but has not yet been bound to a local protocol address. |
| T_UNINIT | 0 | The endpoint is uninitialized. This value is returned whenever the system has closed an endpoint, but the client has not (For instance, when a Macintosh goes to sleep, most client providers are closed. Clients are notified with a `kOTProviderWillClose` event, and if they don't close the provider, the system will do it for them). |

# Appendix C - Event Codes

This appendix describes the codes that the Open Transport Library can send to a client's notification routine. For more information about what your client should do when its notification routine receives a particular event code, refer to the section Event Handling.

**Table C-1.** Open Transport Event Codes

| Event code | Hexadecimal value | Meaning |
|---|---|---|
| T_ACCEPTCOMPLETE | $20000003 | An `Accept` function has completed. The `cookie` parameter is a pointer to the receiving endpoint for the endpoint that issued the `Accept` function. If the receiving endpoint is different than the one that issued the `Accept` function, it will receive a `NULL` in the `cookie` parameter. |
| T_BINDCOMPLETE | $20000001 | A `Bind` function has completed, and the cookie parameter is the retAddr parameter of the `Bind` call. |
| T_CONNECT | $2 | An incoming connect response to a client initiated connection has been received. Use `RcvConnect` to receive it. The cookie parameter to the notification routine is the sndCall parameter of the client passed to the `Connect` call. |
| T_DATA | $4 | Incoming data has arrived. Use `Rcv` or `RcvUData` to receive it. Another T_DATA event will not be generating until Rcv or RcvUData has been called and returns a kOTNoDataErr error. |
| T_DELNAMECOMPLETE | $2000000E | A `DeleteName` function has completed. The cookie parameter is either the `OTNameID` (For `OTDeleteNameByID`) or it is the `TNetbuf` pointer that contained the name to delete (for `OTDeleteName`). |
| T_DISCONNECT | $10 | A connection has been torn down. Also used to indicate a client initiated connect has been denied by the remote endpoint. Use `RcvDisconnect` to clear the event. The cookie parameter to the notifier is `NULL` for a T_DISCONNECT event that indicates an established connection has been torn down. If the T_DISCONNECT event indicates a rejected connection request, then the cookie parameter to the notification routine is the same as the sndCall parameter that the client passed to the `Connect` call. |






| | | |
|---|---|---|
| T_DISCONNECTCOMPLETE | $20000005 | A `SndDisconnect` function has completed, the cookie parameter is the call parameter of `SndDisconnect` |
| T_EXDATA | $8 | Incoming expedited data has arrived. Use `Rcv` to receive it. Another T_EXDATA event will not be generating until Rcv has been called and returned either a kOTNoDataErr error, or returned normal (non-expedited) data. |
| T_GETINFOCOMPLETE | $2000000A | A `GetEndpointInfo` function has completed, and the cookie parameter to the notification routine is the info parameter of `GetEndpointInfo` |
| T_GETPROTADDRCOMPLETE | $20000008 | A `GetProtAddress` function has completed, and the `cookie` parameter to the notifier is the `peerAddr` parameter that the client passed into the `GetProtAddress` call. If the client passed `NULL` as the `peerAddr` parameter, then the client's `boundAddr` parameter is passed as the `cookie`. |
| T_GODATA | $100 | Flow control restrictions have been lifted |
| T_GOEXDATA | $200 | Flow control restrictions on the expedited data channel have been lifted. |
| T_LISTEN | $1 | An incoming connection request has arrived. Call `Listen` to receive it. |
| T_LKUPNAMECOMPLETE | $2000000F | A `LookupName` function has completed. The `cookie` parameter is the `TLookupReply` pointer passed in the `LookupName` call. |
| T_LKUPNAMERESULT | $20000010 | A `LookupName` function has found a name and is returning it, but the lookup is not completed yet. The `cookie` parameter is the `TLookupReply` pointer passed in the `LookupName` call. |
| T_MEMORYRELEASED | $2000000C | This event is only used when a client has issued the AckSends function to an endpoint and the endpoint is asynchronous. The event occurs when any sending function has completed and is done using the client memory. The cookie parameter is the `buf` parameter of `Snd` or the appropriate ".buf" parameter of `the structure used to` |

| | | |
|---|---|---|
| | | send data, and the result parameter is the length of that buffer (or the number of bytes that were sent from that buffer in the case of a send that was incomplete due to flow control). |
| T_OPENCOMPLETE | $20000007 | An asynchronous open call is complete. The cookie parameter will be set to the appropriate ProviderRef if no error occurred. |
| T_OPTMGMTCOMPLETE | $20000006 | An OptionManagement function has completed, and the cookie parameter to the notification routine is the ret parameter that the client passed to the OptionManagement function. If the client specified NULL as the ret parameter, then the client's req parameter is passed as the cookie. |
| T_ORDREL | $80 | The remote client has called SndOrderlyDisconnect; your client should now call RcvOrderlyDisconnect. |
| T_REGNAMECOMPLETE | $2000000D | An RegisterName function has completed. The cookie parameter is the TRegisterReply* parameter, unless it was NULL. Then it is the TRegisterRequest* parameter. |
| T_REPLY | $0800 | An incoming response to an outstanding request has been received. Use the RcvUReply or RcvReply function to read the reply. Another T_REPLY event will not be generating until RcvReply or RcvUReply has been called and returns a kOTNoDataErr error. |
| T_REPLYCOMPLETE | $20000004 | A SndUReply or SndReply function has completed, and the cookie parameter is the sequence number of the original request. |
| T_REQUEST | $400 | An incoming request has been received. Use RcvRequest or RcvURequest to receive it. Another T_REQUEST event will not be generating until RcvRequest or RcvURequest has been called and returns a kOTNoDataErr error. |
| T_RESET | $2000 | A connection-oriented endpoint has received a reset from the remote end and has flushed all unread and unsent data. This only occurs for some types of endpoints, and |






| | | |
|---|---|---|
| | | generally leaves the endpoint in an unknown state. |
| T_RESOLVEADDRCOMPLETE | $20000009 | A ResolveAddress function has completed, and the cookie parameter is the retAddr parameter of ResolveAddress |
| T_UDERR | $40 | An SndUData function has failed after previously completing with no error. |
| T_UNBINDCOMPLETE | $20000002 | An Unbind function has completed. The cookie parameter is NULL. |

# Appendix D - Result Codes

This appendix describes the result codes that Open Transport preferred-C routines return. For information about what your client should do when a particular function returns a particular code, refer to the description of that function. For information about XTI result codes, refer to the X/Open Transport Interface specification.

**Table D-1.** Open Transport Preferred Result Codes corresponding to XTI result codes

| Result code | Value | Meaning |
|---|---|---|
| kOTAccessErr | -3152 | The user does not have permission to negotiate the specified address or options. |
| kOTAddressBusyErr | -3172 | The requested address is in use, or this endpoint does not support multiple connections with the same local and remote addresses. This result code indicates that a connection already exists. As a return value for a Bind call, it may also indicate that no dynamic addresses are available for protocols or configuration methods that allow dynamic addressing. |
| kOTBadAddressErr | -3150 | The specified protocol address was in an incorrect format or contained illegal information. |
| kOTBadDataErr | -3159 | The amount of client data specified was not within the bounds allowed by the endpoint. |
| kOTBadFlagErr | -3165 | An invalid flag was specified. |
| kOTBadNameErr | -3170 | The endpoint name is invalid. |
| kOTBadOptionErr | -3151 | The specified protocol options were in an incorrect format or contained illegal information. |
| kOTBadQLenErr | -3171 | The argument qlen when the endpoint was bound with Bind was zero. |
| kOTBadReferenceErr | -3153 | The specified EndpointRef or TEndpoint* does not refer to a valid endpoint. |
| kOTBadSequenceErr | -3156 | An invalid sequence number was specified, or a NULL call pointer was specified when rejecting a connection request. |
| kOTBadSyncErr | -3179 | A call to Sync was made at non-SystemTask time. |
| kOTBufferOverflowErr | -3160 | The number of bytes allocated to hold a result is greater than zero, but not sufficient to store the result. |
| kOTCanceledErr | -3180 | An outstanding call was canceled. |






**Table D-1 (cont').** Open Transport Preferred Result Codes

| | | |
|---|---|---|
| kOTFlowErr | -3161 | The endpoint is in asynchronous mode, but the flow control mechanism prevents the endpoint from accepting any data at this time. |
| kOTIndOutErr | -3173 | There are outstanding connection indications on the endpoint. All other connection indications must be handled either by rejecting them with SndDisconnect, or by accepting them with Accept. |
| kOTLookErr | -3158 | An asynchronous event has occurred on this endpoint. |
| kOTNoAddressErr | -3154 | The endpoint could not allocate an address, or an address was required and not supplied by the client. |
| kOTNoDataErr | -3162 | This endpoint is in non-blocking mode, but no data is currently available. It is also returned by LookupName when no names are found. |
| kOTNoError | 0000 | The function completed execution without error. |
| kOTNoDisconnectErr | -3163 | No disconnect indication is available. |
| kOTNoReleaseErr | -3166 | No orderly release indication currently exists on this endpoint. |
| kOTStructureTypeErr | -3169 | An unsupported structure type was passed in the structType field. This error is also returned when the structType field is inconsistent with the endpoint type. |
| kOTNotSupportedErr | -3167 | This action is not supported by this endpoint. |
| kOTNoUDErrErr | -3164 | No unit data error indication currently exists on this endpoint. |
| kOTOutStateErr | -3155 | The function was issued in the wrong sequence. |
| kOTProviderMismatchErr | -3174 | The endpoint that is to accept the connection is not the same kind of endpoint as this one. |
| kOTQFullErr | -3177 | The maximum number of outstanding indications has been reached for the endpoint. |
| kOTProtocolErr | -3178 | An unspecified protocol error occurred. |

**Table D-1 (cont').** Open Transport Preferred Result Codes

| | | |
|---|---|---|
| kOTResAddressErr | -3176 | The address to which this endpoint is bound differs from that of the endpoint that received the connection request; thus, this endpoint cannot accept this connection request. |
| kOTResQLenErr | -3175 | When this endpoint was bound (see Bind), the qlen parameter was greater than zero. But to accept a connection on an alternate end-point, such as this one, the endpoint must be bound with a qlen parameter equal to zero. |
| kOTStateChangeErr | -3168 | The endpoint is undergoing a transient state change. This error is returned when a function call is made while an endpoint is in the process of changing states. The client should wait for an event indicating the endpoint has finished changing state and call the function again. (Note that the equivalent state-change error code, TSTATECHNG, is not described in the 1992 X/Open XTI specification.) This error is also returned if you attempt to use an "incompatible" function while another operation is still ongoing (e.g. calling SndUData while an OptionManagement call is still outstanding). |






**Table D-2** Open Transport Preferred Result Codes corresponding to UNIX result codes

| Result code | Value | Meaning |
|---|---|---|
| kENOENTErr | -3201 | This error literally means "no such file or directory". In XTI (and Open Transport) it is returned when an attempt is made to open an endpoint or mapper that does not exist in the system. |
| kENIOErr | -3204 | An I/O error occurred. |
| kENXIOErr | -3205 | ????. |
| kENOMEMErr | -3211 | Open Transport cannot allocate enough memory to meet your request. |
| kEBUSYErr | -3215 | The device you are trying to access is busy and could not complete your request. |
| kEINVALErr | -3221 | ???? |
| kEWOULDBLOCKErr | -3234 | In order to complete the operation the request, Open Transport would have to block, and the endpoint is in non-blocking mode. |
| kETIMEDOUTErr | -3259 | The requested operation timed out. |
| kENOSRErr | -3271 | Open Transport cannot allocate enough system resources (usually STREAMS messages) to meet your request. |

# Appendix E - Open Transport and XTI

This appendix describes how Open Transport differs from XTI. Also, it describes which entities in the preferred-C interface of Open Transport corresponds to which XTI entities, and vice versa.

**CAUTION:** The preferred-C interface of Open Transport is based on XTI but is not identical with it. As a result, some elements have no XTI counterparts, and those that have counterparts are not necessarily identical with them. For definitive information about XTI, refer to the X/Open Transport Interface specification.

## Function Names

**Table E-1.** XTI-to-Open Transport Function Cross-Reference

| XTI Function | Open Transport Function |
|---|---|
| t_accept | Accept |
| t_alloc | Alloc |
| t_bind | Bind |
| t_close | CloseProvider |
| t_connect | Connect |
| t_error | — |
| t_free | Free |
| t_getprotaddr | GetProtAddress |
| t_getinfo | GetEndpointInfo |
| t_getstate | GetEndpointState |
| t_listen | Listen |
| t_look | Look |
| t_open | OpenEndpoint |
| t_optmgmt | OptionManagement |
| t_rcv | Rcv |
| t_rcvconnect | RcvConnect |






**Table E-1.** XTI-to-Open Transport Function Cross-Reference (cont')

| XTI Function | Open Transport Function |
|---|---|
| t_rcvdis | RcvDisconnect |
| t_rcvrel | RcvOrderlyDisconnect |
| t_rcvudata | RcvUData |
| t_rcvuderr | RcvUDErr |
| t_snd | Snd |
| t_snddis | SndDisconnect |
| t_sndrel | SndOrderlyDisconnect |
| t_sndudata | SndUData |
| t_strerror | — |
| t_sync | Sync |
| t_unbind | Unbind |

Table E-2 describes shows which names in the Open Transport preferred-C interface correspond to which names in the XTI-style interface.

**Table E-2.** Open Transport-to-XTI Function Cross-Reference

| Open Transport Function | XTI Function |
|---|---|
| Accept | t_accept |
| AckSends | — |
| Alloc | t_alloc |
| Bind | t_bind |
| CloseProvider | t_close |
| Connect | t_connect |
| DontAckSends | — |
| Free | t_free |

**Table E-2.** Open Transport-to-XTI Function Cross-Reference (cont')

| Open Transport Function | XTI Function |
|---|---|
| GetEndpointInfo | t_getinfo |
| GetProtAddress | t_getprotaddr |
| GetNotifier | — |
| GetEndpointState | t_getstate |
| InstallNotifier | – |
| IsNonBlocking | – |
| IsSynchronous | – |
| Listen | t_listen |
| Look | t_look |
| OpenEndpoint | t_open |
| OptionManagement | t_optmgmt |
| Rcv | t_rcv |
| RcvConnect | t_rcvconnect |
| RcvDisconnect | t_rcvdis |
| RcvOrderlyDisconnect | t_rcvrel |
| RcvRequest | – |
| RcvUData | t_rcvudata |
| RcvUDErr | t_rcvuderr |
| RcvURequest | – |
| RemoveNotifier | – |
| ResolveAddress | – |
| SetAsynchronous | – |
| SetBlocking | – |
| SetNonBlocking | – |
| SetSynchronous | – |






**Table E-2.** Open Transport-to-XTI Function Cross-Reference (cont')

| Open Transport Function | XTI Function |
|---|---|
| Snd | t_snd |
| SndDisconnect | t_snddis |
| SndOrderlyDisconnect | t_sndrel |
| SndReply | – |
| SndRequest | – |
| SndUData | t_sndudata |
| SndUReply | – |
| SndURequest | – |
| Sync | t_sync |
| Unbind | t_unbind |

# Extensions to XTI

Table E-3 lists the Open Transport routines that are not part of XTI. Although this document refers to these routines by their Open Transport preferred-C names, client can also call these routines by the XTI-style the names listed in the table.

**Table E-3.** Open Transport Functions not found in XTI

| Open Transport Preferred-C Name | XTI-Style Name |
|---|---|
| *AckSends* | — |
| *DontAckSends* | — |
| *GetProtAddress* | *t_getprotaddr* |
| *InstallNotifier* | *t_installnotifier* |
| *IsNonBlocking* | *t_isnonblocking* |
| *IsSynchronous* | *t_issynchronous* |
| *RcvRequest* | *t_rcvrequest* |
| *RcvURequest* | *t_rcvurequest* |
| *RemoveNotifier* | *t_removenotifier* |
| *ResolveAddress* | *t_resolveaddr* |
| *SetAsynchronous* | *t_asynchronous* |
| *SetSynchronous* | *t_synchronous* |
| *SndReply* | *t_sndreply* |
| *SndRequest* | *t_sndrequest* |
| *SndUReply* | *t_sndureply* |
| *SndURequest* | *t_sndurequest* |






# Data Structures

Many of the Open Transport functions take pointers to data structures as parameters. The table below shows the standard XTI data structure names and the corresponding preferred interface structure names.

**Table E-4.** XTI-to-Open Transport Data Structure Cross-Reference

| XTI Name | Open Transport Name |
|---|---|
| `int fd` | `EndpointRef fd` |
| `t_info` | `TEndpointInfo` |
| `t_netbuf` | `TNetbuf` |
| `t_bind` | `TBind` |
| `t_discon` | `TDiscon` |
| `t_call` | `TCall` |
| `t_unitdata` | `TUnitData` |
| `t_uderr` | `TUDErr` |
| `t_optmgmt` | `TOptMgmt` |

# Result Codes

When an XTI style function fails, it returns -1 to indicate an error has occurred, and the error is stored in a global variable t_errno. If the value of the error is TSYSERR, then the actual error can be found in the global variable errno. The XTI error numbers are small positive integers.

When an Open Transport preferred function fails, the error code is returned as the result of the function. No global variables are used and all errors are negative numbers.

XTI error codes are small positive numbers with defined constants for each that look like TBADADDR or TFLOW. The Open Transport error codes are negative numbers for consistency with the Macintosh Toolbox, and they have names like `kOTBadAddressErr` and `kOTFlowErr`. There is a corresponding Open Transport error code for every XTI error code.

Table E-5 shows the mapping of XTI error names onto Open Transport error names.

**Table E-5.** XTI-to-Open Transport Result Code Cross-Reference

| XTI Result Code | Open Transport Result Code |
|---|---|
| TACCES | kOTAccessErr |
| TADDRBUSY | kOTAddressBusyErr |
| TBADADDR | kOTBadAddressErr |
| TBADDATA | kOTBadDataErr |
| TBADF | kOTBadReferenceErr |
| TBADFLAG | kOTBadFlagErr |
| TBADNAME | kOTBadNameErr |
| TBADOPT | kOTBadOptionErr |
| TBADQLEN | kOTBadQLenErr |
| TBADSEQ | kOTBadSequenceErr |
| TBADSYNC | kOTBadSyncErr |
| TBUFOVFLW | kOTBufferOverflowErr |
| TCANCELED | kOTCanceledErr |
| TFLOW | kOTFlowErr |
| TINDOUT | kOTIndOutErr |
| TLOOK | kOTLookErr |
| TNOADDR | kOTNoAddressErr |
| TNODATA | kOTNoDataErr |
| TNODIS | kOTNoDisconnectErr |
| TNOREL | kOTNoReleaseErr |
| TNOSTRUCTYPE | kOTStructureTypeErr |
| TNOTSUPPORT | kOTNotSupportedErr |
| TNOUDERR | kOTNoUDErrErr |
| TOUTSTATE | kOTOutStateErr |
| TPROTO | — |
| TPROVMISMATCH | kOTProviderMismatchErr |






| | |
|---|---|
| TQFULL | kOTQFullErr |
| TRESADDR | kOTResAddressErr |
| TRESQLEN | kOTResQLenErr |
| TSTATECHNG | kOTStateChangeErr |
| TSYSERR | - |

# Index