INSIDE MACINTOSH

Open Transport

WWDC Release

May 1996

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

About This Book

This book, *Inside Macintosh: Open Transport,* describes the 1.1 release of the Open Transport networking system, which is a communications architecture that can be used to implement any number of networking and other communications systems. This book discusses only the implementation of Open Transport 1.1 on Apple Macintosh computers. Open Transport provides a set of programming interfaces for applications and processes running on Macintosh computers.

Note
All of the Open Transport 1.1 programming interfaces described in this book are compatible with the Mac OS 8 environment. ◆

To get the most out of this book, read the chapters that cover general Open Transport concepts first. If you are planning to use an AppleTalk or TCP/IP protocol, read the protocol-specific chapters after you are familiar with Open Transport's architecture and general functions. The book is organized with the more general Open Transport concepts covered in the first seven chapters, with the more specific material in the later chapters of the book.

It is best to begin by reading the introductory chapter, "Introduction to Open Transport," because it introduces many terms that are used throughout the rest of this book. This chapter also gives an overview of the Open Transport architecture and the way it is used to implement networking protocols.

The chapter "Providers," describes the generic Open Transport functions that you can use with any provider. The chapters "Endpoints" and "Mappers" introduce functions that are particular for endpoint and mapper providers. The next three chapters, "Option Management," "Configuration Management,"and "Process Management" continue the general discussion of Open Tranport concepts.

The chapter "TCP/IP Services " and the seven AppleTalk-specific chapters describe how to use the Open Transport implementations of AppleTalk and TCP/IP. The last chapter, "Serial Endpoint Providers," describes how to use Open Transport with a serial driver.

At the end of this book are two appendixes: "Open Transport and XTI" and "OT Result Codes."

- "Open Transport and XTI." This appendix describes the correspondence between the XTI and Open Transport client programming interfaces. Open Transport is a superset of XTI and therefore includes functions that are not defined in XTI. This appendix focuses on how general provider functions and endpoint functions correspond to XTI functions.
- "Result Codes." This appendix lists the result codes returned by the Open Transport preferred-C functions.

If you are new to programming for the Macintosh, you can read the book *Inside Macintosh:Overview* for a general introduction to general concepts of Macintosh programming. Other books in the *Inside Macintosh* series are helpful for specific information about other aspects of the Macintosh Toolbox and the Macintosh Operating System. In particular, to benefit most from this book, you should already be familiar with the run-time environment of Macintosh applications, as described in the two books *Inside Macintosh: Processes* and *Inside Macintosh: Memory.*

The information in this book constitutes only a part of the body of literature documenting the AppleTalk and TCP/IP protocol families and the XTI standard upon which Open Transport is based.

For more information about the AppleTalk protocol family, see the book *Inside AppleTalk*, second edition, which has detailed specifications for each of the AppleTalk protocols.

For more information about the TCP/IP protocol family, see any good book on TCP/IP. Two such books for information on TCP/IP protocol internals are *TCP/IP Illustrated, Volume 1* by W. Richard Stevens and *Internetworking with TCP/IP, Volume 1* by Douglas E. Comer.

For more information about the XTI standard, see *X/Open CAE Specification (1992): X/Open Transport Interface (XTI)*. The Open Transport TCP/IP software modules are based on the UNIX Streams architecture. For more information about Streams, see *UNIX System V Release 4: Programmer's Guide: STREAMS*.

Format of a Typical Chapter

Most of the chapters in this book follow a standard structure. For example, the chapter "Endpoints" contains these sections:

- "About Endpoints." This section presents a general introduction to endpoints and endpoint providers.
- "Using Endpoints." This section provides an overview of the features provided by Open Transport for endpoints.
- "Endpoints Reference." This section provides a complete reference for the endpoints and endpoint providers by describing the data types, constants, and functions they use. Each function description also follows a standard format, which presents the function declaration followed by a description of each of its parameters.

The chapters that cover AppleTalk and TCP/IP protocols include a subsection that describes the protocol-specific information for certain general Open Transport functions. For example, the chapter "TCP/IP Services" includes the following section:

- "Using General Open Transport Functions With TCP/IP." This section describes any special considerations that must be taken into account for general endpoint and mapper Open Transport functions when using them with the Open Transport TCP/IP implementation.

Conventions Used in This Book

Inside Macintosh uses special conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and names of actual data structures, fields, constants, parameters, and routines are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but not essential to an understanding of the main text. (An example appears in the chapter "Introduction to Open Transport" on (page 1-6).) ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears in the chapter "Endpoints" on (page 3-87).) ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears in the chapter "Endpoints" on (page 3-63).) ▲

The Development Environment

The Open Transport functions described in this book are available using C or C++ language interfaces. How you access these functions depends on the development environment you are using.

All code listings in this book are shown in ANSI C. They show ways of using various functions and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and in many cases tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application.

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone:	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (elsewhere in the world)
Fax:	716-871-6511
AppleLink:	APDA
America Online:	APDAorder
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-2T
Cupertino, CA 95014-6299

Introduction to Open Transport

Contents

This chapter provides an overview of the Open Transport networking system. Open Transport is a communications architecture that can be used to implement any number of networking and other communications systems. This book discusses only the implementation of Open Transport on Apple Macintosh computers. Open Transport provides a set of programming interfaces for applications and processes running on Macintosh computers.

This chapter introduces some of the terminology that is used throughout the rest of this book. Read this chapter to gain an overview of the Open Transport architecture and the way it's used to implement networking protocols. You should also read this chapter for suggestions on which networking protocols to use for various application requirements.

This chapter begins with a brief description of Open Transport and the advantages it provides over earlier Macintosh networking architectures. Next, "Basic Networking Concepts" defines a variety of terms used in Open Transport and in networking in general. The section "About Networking With Open Transport" describes the Open Transport architecture and describes some concepts important to Open Transport: providers, transport independence, and endpoints. Finally, the section "Deciding Which Protocol to Use" gives you guidelines to help you decide which protocol or protocol family to use for a given purpose.

The chapters that make up the rest of this book describe how to use the Open Transport programming interface and the Open Transport implementations of AppleTalk and TCP/IP.

Introduction to Open Transport

Open Transport is the networking architecture used by Apple Computer, Inc. for Macintosh computers. Whereas AppleTalk provided a proprietary networking system for Macintosh computers, the current Macintosh Operating System with Open Transport provides not only AppleTalk but also the industry-standard TCP/IP protocols and serial connections. In addition, the Open Transport architecture allows developers to add other networking systems to the Macintosh Operating System without altering the user's experience or the application programming interface (API).

The independence of the APIs from the underlying networking or transport technology is called **transport independence** and is one of the cardinal features of Open Transport. Whereas the APIs are independent of the networking system in use, the specific set of functions you call does depend on the nature of the protocols. For example, you use different functions for a connection-oriented protocol like AppleTalk Data Stream Protocol (ADSP) than for a connectionless protocol like Datanet Delivery Protocol (DDP) or Internet Protocol (IP). Transport independence is described in more detail in "Transport Independence" on page 1-20.

Other important features of Open Transport are its support of multihoming and multinodes.

Multihoming allows multiple Ethernet, token ring, FDDI, and other network interface controller (NIC) cards to be active on a single node at the same time. In addition to selecting the type of network connection, the user can select a particular device to be used for the network connection.

Multinode architecture is an AppleTalk feature that allows an application to acquire node IDs that are additional to the standard node ID that is assigned to the system when the node joins an AppleTalk network. Multinode architecture is provided to meet the needs of special-purpose applications that receive and process AppleTalk packets in a custom manner instead of passing them directly on to a higher-level AppleTalk protocol for processing. Multinode IDs allow the system that is running your application to appear as multiple nodes on the network. The prime example of a multinode application is Apple Remote Access (ARA). The chapters "AppleTalk Addressing" and "Datagram Delivery Protocol" in this book describe the use of multinodes.

Basic Networking Concepts

Although this book is intended for readers who already have some knowledge of networking fundamentals, many people use slightly different definitions for the same networking terms. Therefore, this section provides definitions of networking and communications terms as used in this book; the following section, "About Networking With Open Transport," discusses concepts specific to Open Transport.

A **network** is a system of computers and other devices (such as printers and modems) that are connected in such a way that they can exchange data.

A networking system consists of hardware and software. Hardware on a network includes physical devices such as Macintosh personal computer workstations, printers, and Macintosh computers acting as file servers, print servers, and routers; these devices are all referred to as **nodes** on the network.

If the nodes are not all connected to a single physical cable, special hardware and software devices must connect the cables in order to forward messages to their destination addresses. A **bridge** is a device that connects networking cables without examining the addresses of messages or making decisions as to the best route for a message to take. By contrast, a **router** contains addressing and routing information that lets it determine from a message's address the most efficient route for the message. A message can be passed from router to router several times before being delivered to its destination.

In order for nodes to exchange data, they must use a common set of rules defining the format of the data and the manner in which it is to be transmitted. A **protocol** is a formalized set of procedural rules for the exchange of information and for the interactions among the network's interconnected nodes. A network software developer implements these rules in software modules that carry out the functions specified by the protocol.

Whereas a router can connect networks only if they use the same protocol and address format, a **gateway** converts addresses and protocols to connect dissimilar networks.

A set of networks connected by routers or gateways is called an **internet.** The term **Internet** (note the capitalization) is often used to refer to the largest worldwide system of networks, also called the **Worldwide Internet.** The basic protocol used to implement the WorldWide Internet is called the *Internet Protocol*, or *IP*. Because the word *internet* is used in several different ways, it is important to note capitalization and context whenever you see this word.

A networking protocol commonly uses the services of another, more fundamental protocol to achieve its ends. For example, the AppleTalk Data Stream Protocol (ADSP) uses the Datagram Delivery Protocol (DDP) to encapsulate the data and deliver it over an AppleTalk network. The protocol that uses the services of an underlying protocol is said to be a **client** of the lower protocol; for example, ADSP is a client of DDP. A set of protocols related in this fashion is called a **protocol stack.** Protocol stacks are described in more detail in "Protocol Stacks and the OSI Model," beginning on page 1-9.

Note
Although it is sometimes important to distinguish between a protocol and the software that implements the protocol, in most cases you can infer which is meant from the context. Accordingly, this book usually uses the term *protocol* rather than the more precise term *protocol implementation* to refer to the Open Transport implementation of a protocol. ◆

Types of Protocols

Networking protocols can be characterized as connectionless or connection-oriented, and as transactionless or transaction-based.

A **connectionless protocol** is one in which a node that wants to communicate with another simply sends a message without first establishing that the receiving node is prepared to receive it. Each message sent must include addressing information so that it can be delivered to its destination.

A **connection-oriented protocol** is one in which two nodes on the network that want to communicate must go through a connection-establishment process called a **handshake.** This involves the exchange of predetermined signals between the nodes in which each end identifies itself to the other. Once a connection is established, the communicating applications or processes on the nodes at either end can send and receive data without having to add addresses to the messages or repeat the handshake process. Connection-oriented protocols provide support for sessions. A **session** is a logical (as opposed to physical) connection between two entities on a network or internet. A session must be set up at the beginning, maintained by the periodic exchange of information, and broken down at the end. All of these services entail overhead compared to a connectionless protocol, for which no connection setup or breakdown is required and for which no session must be maintained.

A connection-oriented session is analogous to a telephone call. The party who initiates the call knows whether the connection is made because someone at the other end of the line either answers or not. As long as the connection is maintained, neither party needs to dial the other telephone number again. A connectionless protocol is analogous to mail. A person sends a letter expecting it will be delivered to its destination. Although the mail usually arrives safely, the sender doesn't know this unless the recipient initiates a response affirming it. Each letter sent by either party requires a complete address.

A **transactionless protocol** defines how the data is to be organized and delivered from one node to another. A connection-oriented transactionless protocol is used to maintain a **symmetrical connection;** that is, one in which both ends have equal control over the communication. Both ends can send and receive data and initiate or terminate the session. A connectionless transactionless protocol sends data in discrete datagrams. A **datagram,** also referred to as a **packet,** is a small unit of data that includes a **header** portion that holds the destination address (and may contain other information, such as a checksum value), and a data portion that holds the message text. A connection-oriented transactionless protocol can send data as a continuous stream of data or, in some cases, as packets.

If both ends of a connection-oriented transactionless data stream session can transmit and receive data simultaneously, the connection is referred to as **full duplex.** If the two sides have to take turns transmitting and receiving, the connection is referred to as **half duplex.**

A **transaction-based protocol** specifies the sequence and some of the content of messages passed between nodes. When using a transaction-based protocol, the application on one node, known as the *requester,* sends a request to the other application, known as the *responder,* to perform a task. The responder completes the task and returns a response that reports the outcome of the task. Once one node has issued a request, the receiving node is constrained to respond in a predefined way. A transaction-based connection is sometimes referred to as an **asymmetrical connection.**

Table 1-1 shows where some Open Transport protocols fit in the protocol-type matrix. A protocol of one type can be a client of a different type. For example, the connection-oriented transaction-based AppleTalk Session Protocol (ASP) is a client of the connectionless transaction-based AppleTalk Transaction Protocol (ATP), which is in turn a client of the connectionless transactionless Datagram Delivery Protocol (DDP).

Table 1-1 The Open Transport protocol matrix and some Open Transport protocols

	Connectionless	**Connection-oriented**
Transactionless	PPP DDP IP UDP	Serial connection ADSP TCP
Transaction-based	ATP	ASP

Addressing and Data Delivery

In order to establish a network connection or to send a message using a connectionless protocol, you must have the address of the destination. Each protocol uses a specific type of address, which might be the same as that used by a lower-level protocol in the protocol stack or might be unique to that protocol. DDP and IP, for example, use addresses sufficient for node-to-node delivery of datagrams, through routers if necessary. The protocols and applications that are clients of DDP are assigned socket numbers. A **socket** is a piece of software that serves as an addressable entity on a node. DDP is responsible for delivering a datagram to the correct socket.

Similarly, IP delivers each datagram to a specific client protocol—such as Transaction Control Protocol (TCP) or User Datagram Protocol (UDP)—running on a specific node. The processes running the TCP/IP client protocols are each assigned a port number; the client protocol is responsible for delivering the datagram to the correct port number. Each client of IP running on a socket maintains its own list of port numbers. Whereas AppleTalk normally assigns socket numbers dynamically to a process when it registers itself on the network, the TCP/IP port numbers are preassigned by convention or by previous arrangement between users. For more information about AppleTalk addresses, see the chapter "AppleTalk Addressing" in this book. For more information about TCP/IP addresses, see the chapter "TCP/IP Services" in this book.

Low-level connectionless protocols such as DDP and IP usually provide best-effort delivery of data. **Best-effort delivery** means that the protocol attempts to deliver any packets that meet certain requirements, such as containing a valid destination address, but the protocol does not inform the sender when it is unable to deliver the data, nor does it attempt to recover from error conditions and data loss. Higher-level protocols, on the other hand, can provide reliable delivery of data. **Reliable delivery** includes error checking and recovery from error or loss of data.

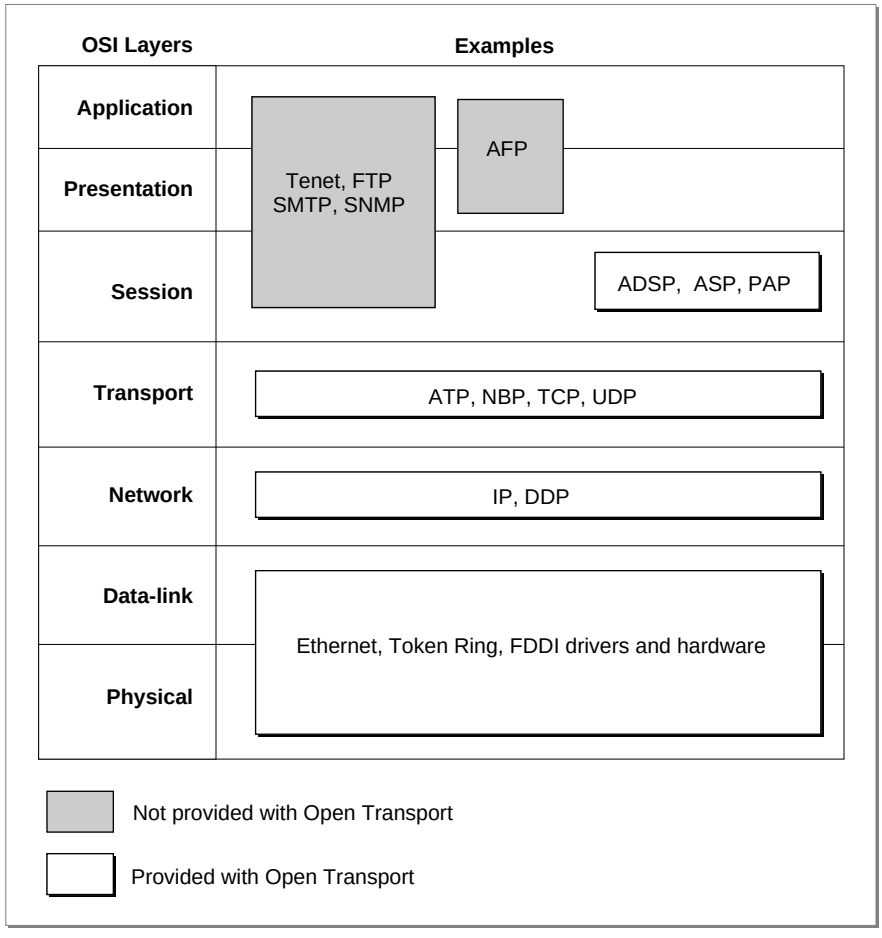
Protocol Stacks and the OSI Model

Most networking systems are designed as layered architectures in which low-level protocols provide services to higher-level protocols in the same protocol stack. Network designers relate each protocol to a reference model, which provides guidelines as to what sort of services should be provided by a protocol at a certain level in the hierarchy. Because these reference models provide a framework that makes it easier to compare the services offered by different protocols, this book shows how each protocol discussed relates to one or more reference models. In this section, the Open Systems Interconnection (OSI) model is described. The OSI model is a seven-layered standard that was published by the International Standards Organization (ISO) in the 1970s. This is the model with which the AppleTalk networking system architecture is most closely aligned.

It is important to note that often more than one protocol is defined and implemented to handle the requirements of a layer in different ways. In addition, some protocols include functions that span more than one layer specified by a model. For example, in favor of efficiency, a network protocol developer may elect to define a single protocol that spans two or more layers of a reference model.

Figure 1-1 shows the layers of the OSI model and how the AppleTalk and TCP/IP protocols provided with the Open Transport system software fit into this model. See the chapter "TCP/IP Services" in this book for a comparison of the OSI and TCP/IP reference models.

Figure 1-1 The OSI model and Open Transport protocols



Each layer of the OSI model has a specific purpose, as follows:

- The highest layer of the OSI model is the **application layer.** This layer allows for the development of application software. Software written at this layer benefits from the services of all the underlying layers.
- The **presentation layer** assumes that an end-to-end path or connection already exists across the network between the two communicating parties,

and it is concerned with the representation of data values for transfer, or the transfer syntax.

- The **session layer** serves as an interface into the transport layer, which is below it. The session layer allows for establishing a session, which is the process of setting up a connection over which a dialog between two applications or processes can occur. Some of the functions that the session layer provides for are flow control, establishment of synchronization points for checks and recovery during file transfer, full-duplex and half-duplex dialogs between processes, and aborts and restarts.
- The **transport layer** isolates some of the physical and functional aspects of a network from the upper three layers. It provides for end-to-end accountability, ensuring that all packets of data sent across the network are received and in the correct order. This is the process that is referred to as *reliable delivery of data,* and it involves providing a means of identifying packet loss and supplying a retransmission mechanism. The transport layer may also provide connection and session management services.
- The **network layer** specifies the network routing of data packets between nodes and the communications between networks, which is referred to as *internetworking.*
- The **data-link layer** and the **physical layer** provide for connectivity. The communication between networked systems can be via a physical cable made of wire or optical fiber, or it can be via infrared or microwave transmission. In addition to these, the hardware can include a network interface controller (NIC), if one is used. The hardware or transport media and the device drivers for the hardware comprise the physical layer.

  The physical hardware provides nodes on a network with a shared data transmission medium called a *data link*. The data-link layer includes both a protocol that specifies the physical aspects of the data link, and the link-access protocol, which handles the logistics of sending the data packet over the transport medium.

About Networking With Open Transport

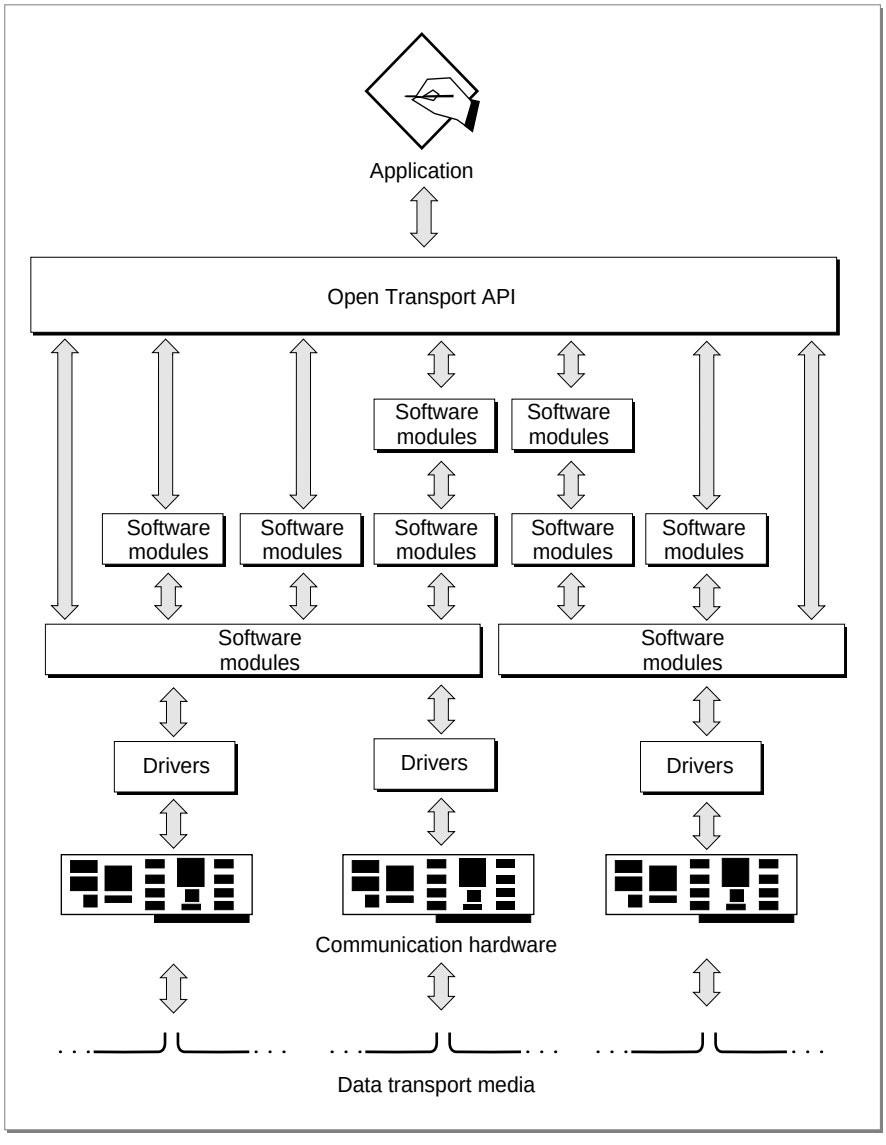
Networking on the Macintosh is implemented through the Open Transport system software. The Open Transport software provides an API that gives you access to the services of the various protocols. The functions you use depend not on the specific protocol you want to use, but on whether the protocol is connection-oriented or connectionless, and whether it is transaction-based or transactionless.

This section describes the architecture of Open Transport and discusses some basic Open Transport features and concepts.

Open Transport Architecture

The Open Transport system software consists of a set of application interface and utility routines (known collectively as the *Open Transport API)*, a set of software modules that implement networking protocols and other services, and hardware drivers. Below the hardware drivers are networking and communications hardware: cards, cables, and built-in ports. These components are illustrated in Figure 1-2 and discussed further in the following sections.

Figure 1-2 The basic architecture of Open Transport



Open Transport API

The Open Transport API consists of two types of functions: utility functions, which are implemented in header files and software libraries; and interface functions, which are implemented by the underlying software modules. Because the interface functions are executed by the software modules, the same function might operate somewhat differently depending on the specific module or modules that executes it. Where such dependencies exist, they are described in the chapter describing a particular protocol.

The Open Transport API is a superset of a standard API defined by the X/Open Company, Ltd. The X/Open API is called the X/Open Transport Interface, or XTI. Both XTI and Open Transport are designed to be independent of the underlying data transport provider; for example, you use the same functions to send a packet of data whether the packet is being transferred by DDP over an AppleTalk network or IP over Ethernet. Whereas XTI specifies functions only for connectionless and connection-oriented protocols, Open Transport also includes functions for transactionless and transaction-based protocols.

The set of functions you use and the sequence of functions you call depends on the operation you want to perform and whether the protocol you want to use is connectionless or connection-oriented, transactionless or transaction-based.

In accordance with XTI, the Open Transport API supports protocol options. An **option** is a value of interest to a specific protocol. For example, an option might enable or disable checksums or specify the priority of a datagram. The available options and their significance are defined by each implementation of each protocol. Every option has a default value, and you can almost always use the default values and not specify any options. It is important to note that, because each option is protocol dependent, specifying a nondefault value for an option decreases or eliminates the transport independence of your application. Protocol options are described throughout this book with the protocol to which they apply. Option handling is described in the chapter "Option Management" in this book.

The XTI specification defines a number of asynchronous events that indicate occurrences such as the arrival of data. Open Transport includes all the standard events defined by XTI, additional asynchronous events, plus completion events that individual functions issue when they complete asynchronous execution. You can poll for asynchronous events, but you cannot

poll for completion events. The preferred method for handling all Open Transport events is to write an event-handling callback function, called a **notifier function.** Open Transport event handling and notifier functions are described in detail in the chapter "Providers" in this book.

Software Modules

The software modules shown in Figure 1-2 on page 1-13 are implemented as Streams modules. The Streams architecture is a UNIX® standard in which protocols (and other service providers) are implemented as software modules that communicate between each other using messages. Open Transport conforms to the Transport Provider Interface (TPI) and Data Link Provider Interface (DLPI) standards, which describe the content and ordering of the messages between modules. In a Streams environment, all modules have the following attributes:

- They process messages asynchronously. One module can send a message to another module and then wait for a reply without interfering with any other system activity.
- They (that is, all the Open Transport Streams modules) share a single address space.
- They may never block; that is, if a module can't complete an operation, it must return with an error rather than indefinitely holding up processing.

Note that Figure 1-2 on page 1-13 shows a very simplified version of the actual Streams architecture. A full AppleTalk or TCP/IP protocol stack has a half-dozen modules that are interconnected.

You can write your own Streams modules to work with Open Transport. The Open Transport TCP/IP software modules are based on the UNIX Streams standard. For more information about Streams, see *UNIX System V Release 4: Programmer's Guide: STREAMS*.

Drivers and Hardware

The Open Transport Streams modules communicate with hardware drivers, which in turn control the flow of data through communications cards or built-in ports. Normally, the user selects which card or port to use through the Chooser. Your application can use the default port for a particular protocol or can configure Open Transport to use a specific port.

Open Transport supports multihoming; that is, an individual node can have more than one hardware device (ports or cards) for a given type of transport. For example, a single computer can have two Ethernet cards, and the user can select which card to use.

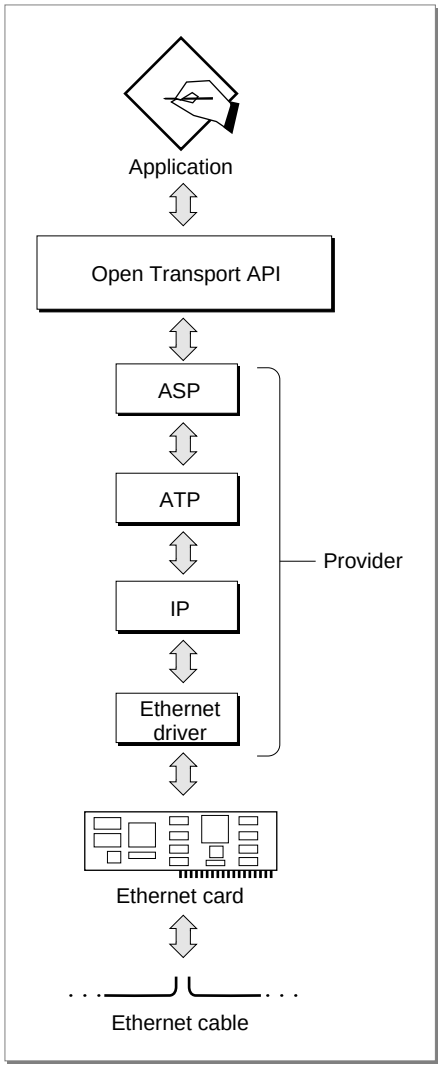
Providers, Endpoints, and Mappers

The concept of a provider is central to an understanding of Open Transport. A **provider** is a set of software modules and drivers that provides a service to clients of Open Transport. For example, when you open an ADSP connection, Open Transport logically links a set of AppleTalk software modules, a communications driver, and a card or port to create what is known as an *ADSP endpoint provider*. The Open Transport software modules implement the Open Transport API, which includes functions for three types of providers:

- endpoint providers
- mapper providers
- service providers

You use an **endpoint provider** to send and receive information over a data link. Figure 1-3 illustrates an ASP endpoint provider.

Figure 1-3 An Open Transport Provider



In order to use an endpoint provider, you must first configure and open an endpoint. An **endpoint** consists of a set of data structures, maintained by Open Transport, that specify the components of the endpoint provider, and the manner in which that provider is to operate (blocking or nonblocking, synchronous or asynchronous, and so forth). An endpoint also maintains state information and other information that Open Transport needs in order to operate that provider.

The Open Transport endpoint functions provide an application programming interface (API) to endpoint providers. When you configure an Open Transport endpoint, you specify which protocol or set of protocols the provider is to use; the highest-level protocol you specify for the endpoint provider determines whether the transport mechanism is connectionless or connection-oriented, and whether it is transactionless or transaction-based. For example, if you specify ADSP as the highest-level protocol in the endpoint provider, the transport is connection-oriented and transactionless.

See "Endpoints and Protocol Layering" on page 1-21 for more information on the configuration of endpoint providers.

You use a **mapper provider** to relate network addresses to network node names and to register and remove node names for networks that support this ability. To use a mapper provider, you must configure and open a **mapper,** a set of data structures that store information about the mapper provider for use by Open Transport.

Mappers implement a standard interface for dealing with addresses. In order to receive data over a network, a process must have a network address. Whereas an address is typically a number of significance to the network software, it is much easier for people using the network to refer to each addressable entity by some name. Consequently, most networks include some naming scheme and a facility that converts between names and addresses. A process using an AppleTalk network must register its name on the network using the Name-Binding Protocol (NBP), which it accesses through a mapper provider.

You use **service providers** to handle features unique to a specific type of Open Transport service. There is no special term used to refer to the data structures maintained by Open Transport for a service provider analogous to an endpoint or mapper.

Because the concept of *zones* is not common to all protocol families, the AppleTalk service provider API includes functions that deal with AppleTalk zones. Similarly, the TCP/IP Domain Service Resolver (DNR) provides some services specific to the TCP/IP protocol family. The TCP/IP service provider functions provide an interface to the DNR.

Each provider supports some subset of the standard Open Transport functions, depending on the nature of that provider; for example, an endpoint provider implements different functions than a mapper provider. What's more, a connection-oriented transactionless endpoint provider implements different functions than a connectionless transaction-based endpoint provider.

Open Transport provides some functions that you can use to control the way a given endpoint or mapper provider operates. For example, you can control whether a provider executes functions synchronously or asynchronously.

When you open an endpoint, mapper, or service , the open function returns a provider reference, analogous to the file reference you get from the File Manager when you open a file. You must specify that provider reference whenever you want to execute a function related to that endpoint, mapper, or service. For example, to send data, you specify the provider reference for the endpoint you want to use.

C++ note

The C++ API for Open Transport includes a class called `TProvider` that is the superclass for all provider-related member functions. Endpoint functions are in class `TEndpoint`, mapper functions are in class `TMapper`, and service provider functions are in classes corresponding to specific protocol stacks. For example, the classes `TAppleTalkServices` and `TInternetServices` contain AppleTalk-specific and TCP/IP-specific member functions.

In object-oriented programming parlance, endpoints, mappers, and the data structures maintained by Open Transport for service providers are all objects. An endpoint, for example, is an object instantiating the class `TEndpoint`. An endpoint contains all the data that Open Transport needs to link together software modules, drivers, and hardware for a specific endpoint provider. All of the Open Transport API functions except the functions that open providers and some utility functions are included in the class definitions of the various classes of providers.

You can call public member functions of the `TProvider` class for provider objects of any type: these functions are the general provider functions. Public member functions defined in a subclass of the `TProvider` class (for example, `TEndpoint`) can be called only for providers belonging to that subclass—in this example, only from the `TEndpoint` subclass. These functions are the type-specific provider functions. Note that, like endpoints and mappers, each kind of service (for AppleTalk, TCP/IP, and so on) derives directly from the `TProvider` class; there is no common class of services. ◆

Transport Independence

In contrast to earlier application programming interfaces (APIs) for AppleTalk, in which each protocol had a separate and unique set of routines, Open Transport provides a single set of functions that you can use with any protocol or protocol family. The type of endpoint you open (connectionless or connection-oriented, and transactionless or transaction-based) determines

which functions you call to send and receive data, independent of the specific protocol or protocol family you use.

For example, if you open a connectionless, transactionless endpoint, you use the `OTSndUData` function to send data. You use this function whether you are using DDP, IP, or UDP. If you open a connection-oriented, transactionless endpoint, on the other hand, you first establish a connection using the `OTConnect` and `OTRcvConnect` functions, and then use the `OTSnd` function to send data. You use these same functions whether you are using TCP, ADSP, or any other Open Transport connection-oriented, transactionless protocol.

You can customize most Open Transport protocols by the specification of option values. Because options are both protocol dependent and implementation dependent, the use of any option values other than the defaults makes your code less transport independent and less portable. Unless you have a compelling reason to change an option value, don't specify any options. You can almost always use the default values provided by Open Transport.

Although transport independence means that you can use the same API regardless of the protocol or communcations hardware you want to use, it does not free you from all knowledge of the transport type. When you open an endpoint, you must specify the highest-level protocol in the endpoint provider, and you must call the functions appropriate to the type of that protocol. For example, although your application can use the same set of functions to send data through either an ADSP or a TCP connection (that is, functions for a connection-based transactionless protocol), you must specify which of these protocols you want to use use when you open the endpoint. Furthermore, to send data using ASP, you must use a different set of functions—for a connection-oriented transaction-based protocol.

Endpoints and Protocol Layering

When you configure an Open Transport endpoint, you specify the highest-level protocol to be used by that endpoint provider. Optionally, you can specify other protocols and ports to be included in the endpoint provider. For example, if you specify only ADSP, Open Transport uses the default underlying protocol for ADSP, which is DDP, over the default AppleTalk port. However, you can specify that ADSP is to use a specific Ethernet card as the port. The endpoint provider consists of the software modules, drivers, and card or internal port that are linked together to provide the service. The services provided by an endpoint provider are an aggregate of the services performed by all the

software modules and hardware pieces that make up the provider. For example, if you specify ADSP to run over DDP through an Ethernet card, you get a session opened and maintained by ADSP, with data encapsulated in DDP packets, converted into digital electronic impulses by the Ethernet card, and transmitted to a DDP address over an Ethernet cable.

Because the type of endpoint you open depends only on the highest-level protocol in the endpoint provider, protocol layering does not affect the transport independence of Open Transport. That is, you use the same functions to open and maintain a connection and to send messages whether you are using ADSP over DDP through Ethernet, or TCP over IP through token ring.

Deciding Which Protocol to Use

Each of the networking protocols available with Open Transport implements a different set of services. This section provides a brief discussion of the uses of each of the protocols included with the Open Transport system software on the Macintosh computer. If you have Open Transport software modules provided by other vendors than Apple Computer, Inc., you should refer to the documentation that came with that software to determine its use.

If you have made provision for the user to select the protocol to be used for communication, you do not need the information in this section. On the other hand, if you are writing an application to perform a specific function, such as to act as a data server, then your choice of protocol or protocols to use depends primarily on your application's needs. In that case, before you open an endpoint, you must make several decisions:

- General purpose or special purpose
- Choice of protocol family
- High-level or low-level protocol
- Connection-oriented or connectionless
- Transaction-based or transactionless

This section discusses each of these choices in turn.

General Purpose or Special Purpose

Your choice of protocol is very simple if there is only one protocol that performs the function you are interested in. For example, if you want to send a print job directly to an AppleTalk printer, you must use the ter Access Protocol (PAP); there is no other choice. On the other hand, if you want to transfer data of a general nature, there are many protocols that can do the job. The following sections describe the factors you can take into consideration to choose among those protocols.

Choice of Protocol Family

There are two sets of protocols, or protocol families, included with the Open Transport system software: AppleTalk and TCP/IP. In addition, other developers can provide protocols and protocol families compatible with Open Transport. You must decide which protocol family to use for a specific purpose. For information on the use of other protocols, see the documentation that came with the software.

AppleTalk is the proprietary networking technology of Apple Computer, Inc. Every Macintosh computer that has ever been made includes AppleTalk hardware and system software. If your application needs to communicate with other Macintosh computers, AppleTalk is a natural choice. Note that the other computers need not be running Open Transport; the nodes must be running the same protocol, but need not be using the same implementation of the protocol.

TCP/IP, on the other hand, is the standard protocol family used by the Worldwide Internet and by many networks owned by businesses and other organizations. Many networking applications running on Macintosh computers that are not using Open Transport cannot communicate over TCP/IP networks. However, if you wish to communicate with the Worldwide Internet without going through a gateway, or if you want to connect to a network that uses TCP/IP protocols, choose one of the Open Transport TCP/IP protocols.

High-Level or Low-Level Protocol

Figure 1-1 on page 1-10 shows the protocols provided by Apple Computer, Inc. with Open Transport and where they fit in the OSI model. The UDP protocol, which is part of the TCP/IP protocol family, is a connectionless transactionless

protocol that provides a minimal amount of error detection in the form of a checksum calculation. If UDP finds that the checksum calculated at any point in the routing process does not match the one calculated when the packet was sent (and stored in the message header), it discards the packet without informing either the sender or receiver of the event.

The other high-level protocols shown in Figure 1-1 provide error checking and error recovery services, including checking for correct packet sequence and retransmission of lost or damaged packets.

If you use a high-level protocol that provides for reliable delivery of data and error recovery, you need not implement these services yourself. On the other hand, these protocols generate somewhat more network traffic than the lower-level protocols, including handshake and control signals, signals to maintain sessions, and retransmitted packets.

The network-layer protocols IP and DDP provide best-effort delivery between nodes on a network. They are connectionless protocols and do not correct for corruption of data, packet loss, or incorrect packet sequencing. They generate the least possible amount of network traffic for the data they transmit. These protocols are appropriate for applications that do not require highly accurate data transmission and for applications that provide their own error recovery. If you want to implement your own protocol stack using an AppleTalk or TCP/IP internet, these are the protocols to use.

Connection-Oriented or Connectionless

Connection-oriented protocols ensure reliable delivery of data and do not require you to repeat the recipient's address or repeat the connection process for the duration of the session. Once you have established a connection, the protocol maintains the connection, informing you if it has closed for any reason. Because of the reliability of connection-oriented protocols, they are a good choice whenever you have a lot of data to exchange over a limited period of time. However, in order to maintain the connection, these protocols sometimes send control signals, which result in increased network traffic.

Open Transport AppleTalk offers three connection-oriented protocols: ADSP, ASP, and PAP. ADSP is a full-duplex transactionless protocol, well suited to the transfer of large amounts of data. ADSP also includes features that let you authenticate the identity of the party at the other end of the connection and send encrypted data, which is then decrypted at the other end. The authentication and encryption features of ADSP are referred to as *AppleTalk Secure Data Stream Protocol (ASDSP)*.

ASP is a transaction-based protocol, best used to implement workstation applications that require an asymmetrical dialog with a server. ASP provides for the setting up, maintaining, and closing down of a session between a workstation and a server. ASP is a client of ATP.

PAP is a transactionless session-layer protocol and a client of ATP. It is intended primarily for communication with Apple Computer's printer products.

Open Transport TCP/IP provides one connection-oriented protocol, TCP, which is a transactionless protocol. TCP, like ADSP, provides highly reliable data delivery suitable for the transfer of large amounts of data.

Transaction-Based or Transactionless

A transaction-based protocol is well suited to many server-client interactions where the client requests services and there are a limited number of ways in which the server can respond. File servers and printers are examples of servers that can use these protocols.

Open Transport AppleTalk includes two transaction-based protocols, ATP and ASP. ATP is connectionless, and ASP is connection-oriented. ASP is a client of ATP.

An ATP transaction request must fit in a single packet; however, the response can contain up to eight packets. ATP transactions are an efficient means of transporting small amounts of data across the network. ATP provides a reliable loss-free transport service.

You should use ATP

- if you want to send a small amount of data
- if your application requires delivery of all packets
- if your application can tolerate a minor degree of performance degradation
- if you do not want to incur the overhead and more extensive performance degradation involved in maintaining a session

A workstation application that requires a state-dependent service should use ADSP or ASP instead of ATP. **State dependence** means that the response to a request is dependent on a previous request. For example, before a workstation application connected to a file server can read a file, it must have first issued a request to open the file. When a dialog is state dependent, all requests must be delivered in order and duplicate packets must not be sent; ADSP and ASP provide for this.

An ATP transaction-based request, such as a workstation application requesting a server to return the time of day, is independent of other requests and not state dependent.

The Open Transport system software provide by Apple Computer, Inc. does not include any transaction-based protocols for the TCP/IP protocol family.

Miscellaneous Events

Open Transport AppleTalk maintains a service called *the miscellaneous events service* that you can use to ensure that your application is not adversely affected when an AppleTalk transition occurs. An example of an AppleTalk transition is an AppleTalk router coming online or a zone name changing. When one of these events occurs, Open Transport sends a message to the notifier functions of all endpoints that have registered for reception of miscellaneous events.

Your application can register itself to receive miscellaneous events by using the `OTIoctl` function, as described in the chapter "Providers" in this book.

Providers

Contents

This chapter describes providers, software entities that offer data-oriented services, and introduces the main types of providers. It also discusses the use of general provider functions, which you can use with any provider regardless of its type. You use these functions to

- open and close providers
- set a provider's mode of operation
- cancel synchronous processing
- issue a command directly to a Streams module underlying a provider

Later chapters in this book describe each type of provider in detail. Although the functions you use to open providers are general provider functions, they are included in the chapter describing individual providers. This chapter describes only the function you use to close a provider because you use the same function for all types of providers.

Before you read this chapter, you should read the chapter "Introduction to Open Transport" in this book. After reading this chapter, you can either read the chapter describing the provider whose services you are interested in. In order to use the functions described in this chapter, you must first use the `OTInitOpenTransport` function to initialize Open Transport. This function is described in the chapter "Configuration Management" in this book.

About Providers

A **provider** is a layered set of protocols, implemented by Streams modules, that provides some kind of data-oriented service. That service might be implementing a networking protocol, encrypting data, filtering data, and so on. When you configure a provider, you can layer the modules that implement the provider to create an arbitrarily complex service for client applications. For example, you can place an encryption module above the AppleTalk Data Stream Protocol (ADSP) module, which is placed above an EtherTalk module. This combination would provide a networking stream of data that was secure from snooping on the network.

Open Transport defines three main kinds of providers:

- endpoint providers
- mapper providers
- service providers

An **endpoint provider** offers a service that creates connections and moves data from one logical address to another. A **mapper provider** offers services that you use to associate, or "map," network entity names with network addresses. A **service provider** lets you perform tasks that are specific to a particular protocol, such as AppleTalk or TCP/IP. There is one type of service provider for each protocol family that Open Transport supports.

In the normal course of events you do not communicate directly with the Streams modules that make up a provider. For example, to use the services of an endpoint provider, you must open an endpoint and use the functions defined in the Open Transport application programming interface (API) for endpoints. The Open Transport API shields your application from the details of the provider implementation, allowing your application to run with little or no change, even when the implementation of the provider is changed, updated, or moved from one platform to another.

To use the services offered by a provider, you must initialize Open Transport and then call the function that opens the provider. When that function returns, it passes back to you a reference to the provider you have just created. A **provider reference** is like a file handle or a driver reference number. It associates a function called from your application with a specific provider that must implement the function; you pass the provider reference as a parameter to all provider functions. The data type of a provider reference depends on the type of the provider (endpoint reference, mapper reference, AppleTalk service reference, and so on).

You can open one provider or many. For example, a server application might open many providers and use them concurrently. The number of providers you can create is limited mainly by the availability of system resources, such as memory. The memory used to create a provider comes partly from your application heap (approximately 8 bytes) but mostly from the system heap. If you allocate data structures while using a provider, the memory for the data structures is allocated entirely from your application heap.

C++ Note
Providers are objects, and each main type of provider is a class. Specifically, endpoints, mappers, and the service providers are all subclasses of the `TProvider` class. Each type of object has a defined C++ interface. ◆

Provider Functions

Functions that manipulate providers are known as **provider functions.** Some provider functions can manipulate providers of any type. These are called **general provider functions** and they are documented in the reference section of this chapter. You use general provider functions to

- get or set a provider's default **mode of operation,** which determines whether provider functions execute synchronously or asynchronously, whether a provider can wait to send or receive data, and whether functions that send data acknowledge having sent the data.
- install and remove a notifier callback function, which the provider uses to pass information to your application
- send a module-specific command, which allows you to communicate directly with the Streams modules that make up your provider
- close a provider

In addition to the general provider functions, each type of provider has type-specific provider functions; these functions work with only that particular type of provider. For example, endpoint functions work only with endpoint providers, and mapper functions work only with mapper providers. Each kind of service provider (for AppleTalk, TCP/IP, and so on) has its own type-specific provider functions. There are no type-specific provider functions that work with more than one type of service provider.

Provider functions that accept a provider reference of type `ProviderRef` are general: they accept any other type of provider reference as well. But functions that require a type of provider reference other than `ProviderRef` (for example, `EndpointRef`) are type-specific: they accept only that type of provider reference.

C++ Note
You can call public member functions of the `TProvider` class from any provider: these functions are the general provider functions. Public member functions defined in a subclass of the `TProvider` class (for example, `TEndpoint`) can be called only from providers belonging to that subclass (in this example, only from the `TEndpoint` subclass): these are the type-specific provider functions. Note that, like endpoints and mappers, each kind of service provider (for AppleTalk, TCP/IP, and so on) derives directly from the `TProvider` class; there is no common class of service providers. ◆

You cannot call most provider functions or other Open Transport functions at interrupt time. You cannot include these functions in any interrupt routine from an external device, VBL task, Time Manager task, or Deferred Task Manager task. You can only call these functions at system task time (primary interrupt level) or at deferred task time (secondary interrupt level) scheduled by the Open Transport function `OTScheduleDeferredTask`. For more information and a list of those functions you can call from an interrupt, see the chapter "Process Management" in this book.

Modes of Operation

For each provider, you can use general provider functions to specify

- how provider functions execute

  In **synchronous mode,** provider functions return only when they complete execution. In **asynchronous mode,** they return as soon as they are queued for execution. Applications running under an operating system that does not use threads, can avoid awkward delays and generally improve performance by calling functions asynchronously.

- the provider's blocking status

  A provider's **blocking status** affects how functions that send and receive data behave when they must wait to complete an operation. If a provider is **blocking,** it waits for as long as it takes to send or receive data. If a provider is **nonblocking,** the provider attempts to send or receive data and, if it cannot do so immediately, it returns with a result indicating why it could not complete the operation.

- the provider's send-acknowledgment status

  A provider's **send-acknowledgment status** determines whether endpoint functions that send data make an internal copy of the data before sending it and whether they advise the provider when the data has actually been sent. Open Transport ignores the send-acknowledgment status for mapper, AppleTalk Services, and TCP/IP providers.

For more information about how you use general provider functions to control a provider's mode of operation, see the section "Controlling a Provider's Modes of Operation" on page 2-8.

Provider Events

Open Transport defines two kinds of events called **provider events.** These events are unique to the Open Transport architecture and not events in the usual Macintosh sense: they are not processed by the Event Manager, and they have no associated event record. Rather, Open Transport uses provider events to inform your application that something has occurred which demands your immediate attention or to signal the fact that a function executing in asynchronous mode has completed. The first kind of event is called an **asynchronous event;** the second kind of event is called a **completion event.** In this book, the term *event* refers to a provider event, except where noted otherwise.

A provider uses asynchronous events to notify your application that data has arrived or that a request for a connection or disconnection is pending. Most asynchronous events defined for Open Transport have equivalents in the X/Open Transport Interface (XTI), from which the Open Transport interface derives. XTI does not define completion events. As just mentioned, a provider uses completion events to notify your application that an asynchronous function has finished executing. Some functions are inherently synchronous and have no corresponding completion event. For example, if an endpoint provider is in asynchronous mode and you execute the `OTGetEndpointState` function, the function returns information about the state of the endpoint immediately. The description of a function indicates whether the function behaves differently in asynchronous mode.

A provider event is identified by a provider event code. These are listed and described in the event codes enumeration beginning on page 2-17. All provider event codes begin with the prefix `T_`, as in `T_DATA`. Provider event codes for completion events end in the suffix `COMPLETE`, as in `T_BINDCOMPLETE`. Codes for asynchronous events have no uniform suffix.

In general, to receive notice of provider events, you must provide a notifier function and install it for the provider. A **notifier function** is a function that you write and that the provider can call when an event occurs. When the provider calls this function, it uses the function's parameters to pass back information about the event that occurred, and if this is a completion event, it also passes back additional information about the result of the function that completed and a pointer to any other information passed back by the function. The section "Using Notifier Functions to Handle Provider Events," beginning on page 2-13 provides additional information about notifier functions and the issues involved in asynchronous processing. You can also refer to "Application-Defined Functions," beginning on page 2-45 for a description of the notifier function.

Using Providers

This section explains how you obtain and change a provider's mode of operation, it introduces the `TNetBuf` structure, which is universally used in Open Transport to transfer data, it provides more detailed discussion of asynchronous processing and the use of notifier functions, and it explains how you close a provider.

In addition to the functions used to set a provider's mode of operation and to close a provider, general provider functions include the `OTIoctl` function, which you can use to communicate directly with a Streams module implementing a networking protocol. For more information, see the description of the function in the reference section to this chapter.

Controlling a Provider's Modes of Operation

A provider's mode of operation determines how provider functions execute and determines the behavior of provider functions that send and receive data. You can control a provider's mode of operation by calling general provider functions to specify whether provider functions execute synchronously or asynchronously, whether provider functions can block, and whether they can acknowledge sends. The following three sections provide additional information about how you can obtain a provider's current mode of operation and how you can change it.

Specifying How Provider Functions Execute

For each provider, you can control whether provider functions run synchronously or asynchronously. When you open a provider, you set its default mode of execution. For example, when you open an endpoint provider, you can use either the function `OTOpenEndpoint` or `OTAsyncOpenEndpoint`. If you open an endpoint provider using the `OTAsyncOpenEndpoint` function, Open Transport creates the provider and sets the default execution mode for all the provider's functions to asynchronous.

A provider's default mode of execution remains in effect until you change it by calling either the `OTSetSynchronous` function or the `OTSetAsynchronous` function. The new mode remains in effect until you change the mode again. A provider's mode of execution affects only that provider. If you use two or more providers, they need not operate in the same mode.

In general, you should use providers in asynchronous mode. Although you can call all of a provider's functions synchronously, doing so generally results in a poor user experience because the user's system can do nothing else while a function is executing. This is especially likely to happen when heavy network traffic prevents a function that is sending or receiving data from completing. However, asynchronous processing does require some additional work: you must make sure that memory you have allocated for a function's output parameters is persistent and you must use some sort of mechanism to determine when the function has actually completed. These issues are taken up in the section "Using Notifier Functions to Handle Provider Events," beginning on page 2-13.

If you plan to call provider functions in synchronous mode, you should avoid doing so when you don't know how long it might take for a function to complete or when the function is being called from a function that executes at interrupt time.

The return behavior of certain provider functions is controlled not only by a provider's mode of execution but also by the provider's blocking status, described in the following section. Changing a provider's mode of execution does not change its blocking status.

Setting a Provider's Blocking Status

A newly created provider does not block, regardless of which Open Transport function created it. After a provider is created, you can change its blocking status as often as you like. A provider's blocking status affects only that provider.

- You use the `OTSetBlocking` function to set a provider's mode of operation to blocking.
- You use the `OTSetNonBlocking` function to set a provider's mode of operation to nonblocking.
- You use the `OTIsNonBlocking` function to determine whether a provider blocks.

If a provider is nonblocking, provider functions that cannot complete send or receive operations return an error indicating why the operation could not complete. The result returned might be

- `kEAGAINErr` or `kEWOULDBLOCKErr`, indicating that the function would have to be queued before it could execute
- `kOTNoDataErr`, indicating that data has not yet arrived
- `kOTFlowErr`, indicating that network traffic is too heavy to allow immediate execution

In all these cases, you should call the function again.

Setting a Provider's Send-Acknowledgment Status

You can control the behavior of provider functions that send data by specifying that a provider acknowledge sends. For now, you can only specify that endpoint providers acknowledge sends.

By default, providers do not acknowledge sends. This means that when you use a function that sends data, the provider copies the data into an internal buffer and then sends the data. Once the provider has copied the data into its own buffer, it releases the buffer you have allocated for the data. As soon as the function returns, you can change the contents of your buffer—even if the provider has not yet sent the data it copied.

If you use the `OTAckSends` function to specify that the endpoint provider acknowledge sends and you call a function that sends data, the endpoint provider does not copy data from your buffer before sending it. Instead it reads

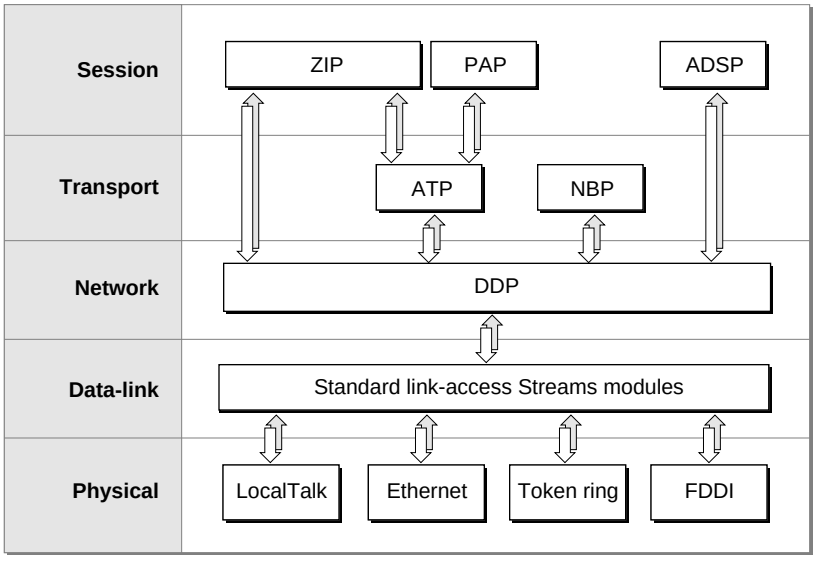
data directly from your buffer while sending. For this reason, you must not change the contents of your buffer until the endpoint provider is no longer using it. Sometimes, particularly if the endpoint is in blocking mode, a send operation can be delayed. The provider lets you know that it has finished using the buffer by calling your notifier function and passing `T_MEMORYRELEASED` for the `code` parameter, a pointer to the buffer that was sent in the `cookie` parameter, and the size of the buffer in the `result` parameter.

Only endpoint provider functions are affected by your calling the `OTAckSends` and `OTDontAckSends` functions. For additional information, see the discussion of an endpoint's mode of operation in the chapter "Endpoints" in this book.

Sending and Receiving Data

Most provider functions that transfer data pass a parameter of type `TNetbuf` that specifies the size and location of user data. Such data is usually an address, option information, or actual data that you want to transfer. You can think of the `TNetbuf` structure as Open Transport's universal bucket, used to hold and pass on different kinds of information. Figure 2-1 shows how the `TNetbuf` structure refers to data in memory.

Figure 2-1 The `TNetbuf` structure



The structure is composed of three fields: the `buf` field, the `len` field, and the `maxlen` field. The `buf` field contains the beginning address of the data; the `len` field specifies the size of the data; and the `maxlen` field specifies the maximum size the data could take up. How you use this structure depends on whether the structure specifies an input or output parameter:

- If you are sending information (the structure is used to specify an input parameter), you must allocate a buffer and initialize it to contain the data you want to send. Then you must set the `buf` field to point to the buffer and set the `len` field to specify the size of the data.
- If you are receiving information (the structure is used to specify an output parameter), you must allocate a buffer into which the function can place the information when it returns. Then you must set the `buf` field to point to the buffer and set the `maxlen` field to specify the maximum size of the data that could be placed in the buffer. When the function returns, it sets the `len` field to the actual size of the data.

If you are using an endpoint provider, Open Transport also allows you to send noncontiguous data. If you need to do this, you use an `OTData` structure to

specify the size and location of the data. For more information, see the chapter "Endpoints" in this book.

If you want to do a no-copy receive (that is, to receive data without doing the usual extra buffer copying involved in receiving data), you use a special `OTBuffer` structure that specifies the size and location of the data. For more information, see the chapter "Endpoints" in this book.

Using Notifier Functions to Handle Provider Events

When provider functions execute asynchronously, you can continue processing without having to wait for a function to complete execution. In some cases, you might need to know when the function has finished executing, either because further processing depends on the results of that operation or because you need to use memory you have allocated for that function. In order to meet this need, the Open Transport architecture defines completion events, which are generated by a provider when an asynchronous function completes execution. To pass the event to your application as well as other information about the function that has completed, the provider calls a notifier function, that you have written and installed for that provider. The provider uses the notifier's parameters to pass the following information back to your application:

- an event code identifying the function that has completed
- the function result
- a pointer to additional information that the function is returning

  This parameter is called the `cookie` parameter. For example, when you call a function that assigns an address to an endpoint, you can request a particular address. When the function returns, it passes back the address that is actually assigned to the endpoint. If you call the function asynchronously, this information is referenced by the `cookie` parameter.

- a context pointer for your use

  You define this pointer when you install the notifier function. When the provider calls the notifier, it passes this pointer back to you.

If you open a provider in asynchronous mode, you install a notifier function by passing a pointer to it in one of the parameters to the function used to open the provider. If you open a provider in synchronous mode, you must call the `OTInstallNotifier` function to install the notifier. To remove a notifier, call the `OTRemoveNotifier` function. If you want to change notifiers, you must call the

`OTRemoveNotifier` function to remove the old notifier, and then call the `OTInstallNotifier` function to install the new notifier.

You are responsible for the contents of a notifier function. Typically, such a function tests to see whether the function that just completed has returned an error. If it has not, it uses a `switch` statement to transfer control to different subroutines, depending on the event code passed to the notifier. Listing 2-1 shows the skeleton of a notifier function used to handle events for an endpoint that is being used to accept connection requests. As you can see, the notifier does not need to handle every completion event, just those that you expect to happen and that have meaning for the provider you are opening. For a more detailed discussion of this code fragment, see the section describing connection-oriented endpoints in the chapter "Endpoints" in this book.

You can use a notifier function to handle asynchronous events as well as completion events. A provider uses asynchronous events to inform your application that data has arrived or that a connection or disconnection request is pending. Endpoint providers have the option of using an endpoint provider function to poll for these events, but all other providers must use the notifier function to respond to asynchronous events. The method used is the same as for completion events. You must include case statements in the notifier that are pertinent to the asynchronous events you expect to receive.

Listing 2-1 A notifier function

```
pascal void MyConnectorEventHandler(EndpointRef *listenEP,
            OTEventCode event, OTResult result, void* cookie)
{

// set the global error result, only if result is negative
    if (result < 0)
        gOTErr = result;
    else
        gOTErr = kOTNoError;

    switch (event)
    {
        case T_OPENCOMPLETE:
        // set flag that the listener endpoint has
        //  completed processing
            gAsyncProcessActive = false;
```

```
            if (result == kOTNoError)
                gListenConnectEP = (EndpointRef)cookie;
            break;

        case T_BINDCOMPLETE:
            break;

        case T_LISTEN:
        // don't expect to get a connect request on this endpoint
            break;
        case T_ACCEPTCOMPLETE:
            break;
        case T_ORDREL:
        case T_DISCONNECT:
        case T_DISCONNECTCOMPLETE:
        // decrement our use counter here
            gListenT_LISTENcnt--;
            break;
        case T_RESET:
            break;

        default:
            break;
    }
}
```

The provider calls your notifier function at deferred task time or at system task time. This means that the routines called from your notifier

- must be reentrant
- cannot move memory
- can’t depend on validity of handles to unlocked blocks
- should not perform time-consuming tasks
- should not be synchronous
- cannot call Open Transport functions in synchronous mode

The only exception to these rules occurs when you are responding to the event `kOTProviderWillClose`. See the event codes enumeration beginning on page 2-17 for additional information.

If you execute provider functions asynchronously, you must also take special care about the duration of the function's variables. A function that is executed asynchronously returns immediately, and the stack frame of the function that called it might be torn down before you have had a chance to retrieve the information returned in the parameters to the asynchronous function (using the notifier function's `cookie` parameter). If these parameters are local variables in the calling function, the information passed back by the asynchronous function is lost. To avoid this situation, you need to write the function that calls the asynchronous function in such a way that the memory pointed to by its parameters is not overwritten. For example, you could make these variables global.

Transferring a Provider's Ownership

Open Transport keeps track of the owner of each provider, and when a client dies or quits without closing all of its outstanding providers, Open Transport attempts to close them on behalf of the client. Every shared library, code resource, or program that creates an endpoint, or uses one of the endpoint functions that allocate memory on behalf of the client, is a client of Open Transport. For ASLM shared libraries and applications, Open Transport can clean up after the library or application easily. For CFM shared libraries, however, the client *must* call `CloseOpenTransport` before terminating (this can be done by making `CloseOpenTransport` the termination procedure for the CFM library).

Although it's not a frequent occurrence, there may be times when it is not convenient for you to lose access to a provider. For example, if you are still using a provider created by a shared library when that shared library is unloaded or you are still using a provider reference passed by another application when that application quits, you will find yourself using invalid references unexpectedly.

In cases where you do not want Open Transport to close a given provider, you can define yourself as its new owner with the `OTTransferProviderOwnership` function (page 2-25). You need to obtain the previous owner's client ID before the client terminates, and then pass it to Open Transport along with the provider reference for the provider. Open Transport allocates a new provider reference and returns the new reference to you. The old provider reference is then obsolete and should not be used.

Closing a Provider

There are two instances in which you need to close a provider:

- when you are through using the services offered by a provider

  You do this by calling the `OTCloseProvider` function and passing the provider reference of the provider you wish to close.

- in response to a `kOTProviderWillClose` event

Closing a provider deletes all memory reserved for it in the system heap, deletes its resources, and cancels any provider functions that are currently executing. If the provider is in asynchronous mode, it is your responsibility to make sure that all outstanding functions have completed before you close the provider.

If you must close the provider in response to a `kOTProviderWillClose` event, note that Open Transport issues this event only at system task time. This means that you can set the endpoint to synchronous mode (from within the notifier function) and call functions synchronously to do whatever clean up is necessary before you return from the notifier.

Providers Reference

This section describes general provider data types and functions, which you can use with providers of any type.

Constants and Data Types

This section describes the constants and data types that you can use with general provider functions.

Event Codes

Your application can include a notifier function that the provider calls to inform you that an asynchronous function has completed or that an asynchronous event has occurred. The provider passes an event code for the function's `code` parameter. The event code specifies the name of the

asynchronous function that has completed or the name of an asynchronous event that has occurred. The provider can also pass information using the `result` and `cookie` parameters to the notifier function. Normally, if the provider calls your notifier because an asynchronous function has completed, the `result` parameter contains the result code for the function and the `cookie` parameter contains additional information whose meaning varies with the function called. For example, if you call the `OTAsyncOpenEndpoint` function, the `cookie` parameter would contain the endpoint reference for the endpoint provider you just opened.

Most of the codes specified by the event codes enumeration are used by endpoint providers and relate to the use of endpoint functions. You might need to read the "Endpoints" chapter in this book to make sense of the following constant name descriptions.

The constant names that the provider can use for the event code are given by the following enumeration:

```
enum {
    T_LISTEN                    = (OTEventCode)0x0001,
    T_CONNECT                   = (OTEventCode)0x0002,
    T_DATA                      = (OTEventCode)0x0004,
    T_EXDATA                    = (OTEventCode)0x0008,
    T_DISCONNECT                = (OTEventCode)0x0010,
    T_ERROR                     = (OTEventCode)0x0020,
    T_UDERR                     = (OTEventCode)0x0040,
    T_ORDREL                    = (OTEventCode)0x0080,
    T_GODATA                    = (OTEventCode)0x0100,
    T_GOEXDATA                  = (OTEventCode)0x0200,
    T_REQUEST                   = (OTEventCode)0x0400,
    T_REPLY                     = (OTEventCode)0x0800,
    T_PASSCON                   = (OTEventCode)0x1000,
    T_RESET                     = (OTEventCode)0x2000,
    T_BINDCOMPLETE              = (OTEventCode)0x20000001,
    T_UNBINDCOMPLETE            = (OTEventCode)0x20000002,
    T_ACCEPTCOMPLETE            = (OTEventCode)0x20000003,
    T_REPLYCOMPLETE             = (OTEventCode)0x20000004,
    T_DISCONNECTCOMPLETE        = (OTEventCode)0x20000005,
    T_OPTMGMTCOMPLETE           = (OTEventCode)0x20000006,
    T_OPENCOMPLETE              = (OTEventCode)0x20000007,
    T_GETPROTADDRCOMPLETE       = (OTEventCode)0x20000008,
    T_RESOLVEADDRCOMPLETE       = (OTEventCode)0x20000009,
```

```
    T_GETINFOCOMPLETE               = (OTEventCode)0x2000000A,
    T_SYNCCOMPLETE                  = (OTEventCode)0x2000000B,
    T_MEMORYRELEASED                = (OTEventCode)0x2000000C,
    T_REGNAMECOMPLETE               = (OTEventCode)0x2000000D,
    T_DELNAMECOMPLETE               = (OTEventCode)0x2000000E,
    T_LKUPNAMECOMPLETE              = (OTEventCode)0x2000000F,
    T_LKUPNAMERESULT                = (OTEventCode)0x20000010,
    kOTProviderIsDisconnected       = (OTEventCode)0x23000001,
    kOTProviderIsReconnected        = (OTEventCode)0x23000002.
    kOTProviderWillClose            = (OTEventCode)0x24000001,
    kOTProviderIsClosed             = (OTEventCode)0x24000002,
    kOTConfigurationChanged         = (OTEventCode)0x26000001,
};
```

Constant descriptions

`T_LISTEN` A connection request has arrived. Call the `OTListen` function to read the request.

`T_CONNECT` The passive peer has accepted a connection that you requested using the `OTConnect` function. Call the `OTRcvConnect` function to retrieve any data or option information that the passive peer has specified when accepting the function or to retrieve the address to which you are actually connected. The `cookie` parameter to the notifier function is the `sndCall` parameter that you specified when calling the `OTConnect` function.

`T_DATA` Normal data has arrived. Depending on the mode of service you are using, you can call the `OTRcvUData` function or the `OTRcv` function to read it. Continue reading data until the function returns with the `kOTNoDataErr` result; you do not get another indication that data has arrived until you have read the entire unit.

`T_EXDATA` Expedited data has arrived. Use the `OTRcv` function to read it. Continue reading data by calling the `OTRcv` function until the function returns with the `kOTNoDataErr` result; you do not get another indication that data has arrived until you have read the entire unit.

`T_DISCONNECT` A connection has been torn down or rejected. Use the `OTRcvDisconnect` function to clear the event.

	If the event is used to signify that a connection has been terminated, the `cookie` parameter to the notifier is `NULL`. If the event indicates a rejected connection request, the `cookie` parameter to the notification routine is the same as the `sndCall` parameter that you passed to the `OTConnect` function.
`T_UDERR`	The provider was not able to send the data you specified using the `OTSndUData` function even though the function returned successfully. You must call the `OTRcvUDErr` function to clear this event and determine why the function failed.
`T_ORDREL`	The remote client has called the `OTSndOrderlyDisconnect` function to initiate an orderly disconnect. You must call the `OTRcvOrderlyDisconnect` function to acknowledge receiving the event and to retrieve any data that might have been sent with the disconnection request.
`T_GODATA`	Flow-control restrictions have been lifted. You can now send normal data.
`T_GOEXDATA`	Flow-control restrictions have been lifted. You can now send expedited data.
`T_REQUEST`	A request has arrived. Depending on the mode of service you are using, you can call the OTRcvRequest function or the OTRcvURequest function to receive it. You must continue to call the function until it returns with the kOTNoDataErr result.
`T_REPLY`	A response to a request has arrived. Depending on the mode of service you are using, you can call the `OTRcvReply` function or `OTRcvUReply` function to receive it. You must continue to call the function until it returns with the `kOTNoDataErr` result.
`T_PASSCON`	When the `OTAccept` function completes, the endpoint provider passes this event to the endpoint receiving the connection (whether that endpoint is the same as or different from the endpoint that calls the `OTAccept` function.) The `cookie` parameter contains the endpoint reference of the endpoint that called the `OTAccept` function.
`T_RESET`	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_BINDCOMPLETE` The `OTBind` function has completed. The `cookie` parameter contains the `retAddr` parameter of the bind call.

`T_UNBINDCOMPLETE` The `OTUnbind` function has completed. The `cookie` parameter is meaningless.

`T_ACCEPTCOMPLETE` The `OTAccept` function has completed. The `cookie` parameter contains the endpoint reference of the endpoint to which you passed off the connection.

`T_REPLYCOMPLETE` The `OTSndUReply` or `OTSndReply` functions have completed. The `cookie` parameter contains the sequence number of the request retrieved with the `OTRcvURequest` or `OTRcvRequest` function.

`T_DISCONNECTCOMPLETE`
The `OTSndDisconnect` function has completed. The `cookie` parameter contains the call parameter of the `OTSndDisconnect` function.

`T_OPTMGMTCOMPLETE` The `OTOptionManagement` function has completed. The `cookie` parameter contains the `ret` parameter that you have passed to the function.

`T_OPENCOMPLETE` An asynchronous call to open a provider has completed. The `cookie` parameter contains the provider reference.

`T_GETPROTADDRCOMPLETE`
The `OTGetProtAddress` function has completed. The `cookie` parameter contains the `peerAddr` parameter that you passed to the `OTGetProtocolAddress` function. If you passed `NULL` for that parameter, the cookie parameter contains the address passed in the `boundAddr` parameter.

`T_RESOLVEADDRCOMPLETE`
The `OTResolveAddress` function has completed. The cookie parameter contains the `retAddr` parameter of the `OTResolveAddress` function.

`T_GETINFOCOMPLETE` The `OTGetEndpointInfo` function has completed. The `cookie` parameter contains the `info` parameter of the `OTGetEndpointInfo` function.

`T_SYNCCOMPLETE` The `OTSync` function has completed. The `cookie` parameter is meaningless.

T_MEMORYRELEASED You are using an asynchronous endpoint that acknowledges sends and an OTSnd or OTSndUData function has completed and is done using the buffers containing the data you are sending. If you have called the OTSnd function, the cookie parameter contains the buf parameter. If you have called the OTSndUData function, the cookie parameter contains the udata parameter. The result parameter contains the number of bytes that were sent. This might be less than the number you meant to send due to flow-control or memory restrictions.

T_REGNAMECOMPLETE The OTRegisterName function has completed. The cookie parameter contains the name parameter of the OTRegisterName function.

T_DELNAMECOMPLETE The OTDeleteName function or the OTDeleteNameByID function has completed. The cookie parameter contains the name parameter or the id parameter of the function, respectively.

T_LKUPNAMECOMPLETE

The OTLookupName function has completed. The cookie parameter contains the reply parameter of the OTLookUpName function.

T_LKUPNAMERESULT An OTLookupName function has found a name and is returning it, but the lookup is not yet complete. The cookie parameter contains the reply parameter passed to the OTLookupName function.

kOTProviderIsDisconnected

Your provider was bound with qlen parameter value greater than 0 and it has been disconnected (is no longer listening). You receive this event after a port has accepted a request to temporarily yield ownership of a port to another provider, which causes this provider to be disconnected from the port in question. This currently only happens with serial ports, but could also happen with other connection-oriented drivers that have characteristics similar to serial ports. You get a kOTProviderIsReconnected message when the port reverts back to this provider's ownership again.

`kOTProviderIsReconnected`

Your provider has been reconnected, that is, the cause for its disconnection has been relieved.

`kOTProviderWillClose`

When you return from the notifier function, Open Transport will close the provider whose reference is contained in the `cookie` parameter. The `result` parameter contains a code specifying the reason why the provider had to close. For example, the user decided to switch links using the control panel.

You can only get this event at system task time. Consequently, you are allowed to set the endpoint to synchronous mode (from within the notifier function) and call functions synchronously before you return from the notifier, at which point, the provider is closed. At this point, any calls other than `OTCloseProvider` will fail with a `kOTOutStateErr`.

`kOTProviderIsClosed`

The provider has 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`. At this point, any calls other than `OTCloseProvider` will fail with a `kOTOutStateErr`.

The TNetbuf Structure

You use a `TNetbuf` structure to specify the location and size of a buffer that contains an address, option information, or user data. Provider functions use `TNetbuf` structures both as input parameters and output parameters. If you are using a `TNetbuf` structure as an input parameter, you use it to specify the location and size of a buffer containing information you want to send. If you are using a `TNetbuf` structure as an output parameter, you use it to specify the location and the maximum size of the buffer used to hold information when the function returns.

The TNetbuf structure is defined by the TNetbuf data type.

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

Field descriptions

maxlen — The size (in bytes) of the buffer to which the buf field points. You must set the maxlen field before passing a TNetbuf structure to a provider function as an output parameter. Open Transport ignores this field if you pass the TNetbuf structure as an input parameter.

len — The actual length (in bytes) of the information in the buffer to which the buf field points. If you are using the TNetbuf structure as an input parameter, you must set this field.

If you pass the TNetbuf structure as an output parameter, on return, the provider function sets this field to the number of bytes the function has actually placed in the buffer referenced by the buf field.

buf — A pointer to a buffer. You must make sure that the buf field points to a valid buffer and that the buffer is large enough to store the information for which it is intended.

Functions

This section describes general provider functions. Before you can use these functions, you must initialize the Open Transport software by calling the InitOpenTransport function, which is described in the chapter "Configuration Management" in this book.

Opening and Closing Providers

To create and open a provider, you use a type-specific provider function—for example, the OTOpenEndpoint or OTAsyncOpenEndpoint function creates and opens an endpoint. These functions are included in the chapters describing individual providers. When you finish using a provider of any type, always call the OTCloseProvider function to close and delete the provider.

OTTransferProviderOwnership

Transfers a provider's ownership to a new client.

C INTERFACE

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

C++ INTERFACE

```
ProviderRef TProvider::OTTransferProviderOwnership(
                    OTClient prevOwner,
                    OSStatus* errPtr);
```

PARAMETERS

`ref`	The provider reference for the provider to be transferred.
`prevOwner`	The client ID of the previous owner.
`errPtr`	A pointer to a result code.

DESCRIPTION

The `OTTransferProviderOwnership` function transfers the ownership of the provider indicated by the `ref` parameter to the current Open Transport client. The previous owner must provide the owner-to-be with its client ID, obtained by using the `OTWhoAmI` function; this is then used by the owner-to-be in the `prevOwner` parameter. Open Transport allocates a new provider reference and returns the new reference as the function result. The old provider reference is then obsolete and should not be used.

SPECIAL CONSIDERATIONS

When installing a notifier into a provider, Open Transport assumes that the `OTNotifyProcPtr` pointer is in the same architecture as the call is being made.

After transferring ownership, remove any already installed notifiers and install your own, unless your architecture is such that a cross-architecture notifier is what you want.

▲ **WARNING**
As long as the client that created the provider remains loaded and is in the same architecture (that is, the PowerPC as opposed to the 68000-family Macintosh CPU environments) as the client using the provider, no damage is done by not making this call. However, if the provider was created under a different architecture than the current client using the provider, attempting to close the provider causes a crash. If you do not use the `OTTransferProviderOwnership` function, it is vital that the provider be closed under the same architecture that opened the provider. ▲

SEE ALSO

To get a client ID, call the `OTWhoAmI` function (page 2-26).

OTWhoAmI

Returns the current client's client ID.

C INTERFACE

```
OSClient OTWhoAmI(void);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

DESCRIPTION

The `OTWhoAmI` function returns the current client's client ID. This function is used by the current owner of a provider that is to be transferred to a new

owner. The current client provides this ID to the new owner for use as the `OTTransferProviderOwnership` function's `prevOwner` parameter.

SEE ALSO

To transfer ownership, use the `OTTransferProviderOwnership` function (page 2-25).

OTCloseProvider

Closes a provider of any type—endpoint, mapper, or service provider.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider to be closed and deleted.

DESCRIPTION

The `OTCloseProvider` function closes the provider that you specify in the `ref` parameter. Closing the provider deletes all memory reserved for it in the system heap, deletes its resources, and cancels any provider functions that are currently executing.

Open Transport does not guarantee that all outstanding functions have completed before it closes the provider. It is ultimately your responsibility to make sure that all provider functions that you care about have finished executing, before you close and delete a provider.

▲ **WARNING**

You need to be sure that are no outstanding `T_MEMORY_RELEASED` events for a provider before you close the provider. Otherwise, Open Transport attempts to deliver the event to a provider that no longer exists, with unpredictable results, such as crashing the system. ▲

SEE ALSO

To create and open an endpoint, call the `OTOpenEndpoint` function or the `OTAsyncOpenEndpoint` function, both described in the chapter "Endpoints" in this book.

To create and open a mapper, call the `OTOpenMapper` function or the `OTAsyncOpenMapper` function, both described in the chapter "Mappers" in this book.

To create and open a AppleTalk service provider, call the `OTOpenAppleTalkServices` function or the `OTAsyncOpenAppleTalkServices` function, both described in the chapter "AppleTalk Services" in this book.

To create and open a TCP/IP service provider, call the `OTOpenInternetServices` function or the `OTAsyncOpenInternetServices` function, both described in the chapter "TCP/IP Services" in this book.

Controlling a Provider's Mode of Operation

A provider's mode of operation determines whether the provider runs synchronously or asynchronously, whether the provider blocks, and whether the provider acknowledges sends.

By default, providers created synchronously operate in synchronous mode; providers created asynchronously operate in asynchronous mode. You can use the `OTSetSynchronous` or `OTSetAsynchronous` function to specify how provider functions should execute. You can use the `OTCanMakeSyncCall` function to find out whether Open Transport permits synchronous calls at a given moment. You can find out a provider's current mode of execution by calling the `OTIsSynchronous` function. If synchronous functions are in progress on a provider, you can cancel all of them by calling the `OTCancelSynchronousCalls` function.

A provider's blocking status governs how provider functions proceed when they cannot read or write data without waiting. If a provider blocks, it waits

until it is able to read or write data, which might require that it wait indefinitely. If a provider does not block, the function used to read or write data returns an error, specifying why it could not complete the operation. You can set a provider's blocking status by calling the `OTSetBlocking` or `OTSetNonBlocking` function. You can find out a provider's current blocking status by calling the `OTIsNonBlocking` function. By default, providers do not block. For more information about blocking, see the section "Setting a Provider's Blocking Status," beginning on page 2-10.

You can use the `OTAckSends` or `OTDontAckSends` function to specify whether a provider acknowledges sends. This determines how a provider handles data that you send and whether it informs you when it has sent the data. To determine whether a provider acknowledges sends, you call the `OTIsAckingSends` function. By default, providers do not acknowledge sends. Mapper and individual service providers like AppleTalk and TCP/IP ignore the setting of this attribute. However, the behavior of endpoint functions that send data is affected by the endpoint provider's acknowledgment status.

OTSetSynchronous

Sets a provider's mode of execution to synchronous.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider whose mode of execution you want to set.

DESCRIPTION

The `OTSetSynchronous` function causes all provider functions to run synchronously when using the provider that you specify.

Changing a provider's mode of execution does not affect its notifier function, if any is installed for this provider; the notifier function remains installed.

SEE ALSO

Modes of execution and notifier functions are described in "Specifying How Provider Functions Execute" on page 2-9.

To set a provider to asynchronous mode, call the `OTSetAsynchronous` function, described next. To find out a provider's mode of execution, call the `OTIsSynchronous` function (page 2-31).

OTSetAsynchronous

Sets a provider's mode of execution to asynchronous.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider whose mode of execution you want to set.

DESCRIPTION

The `OTSetAsynchronous` function causes all functions for the provider specified in the `ref` parameter to run asynchronously. You must install a notifier function

for the provider if it needs to receive completion events. You can install a notifier function either before or after calling the `OTSetAsynchronous` function.

Changing a provider's mode of execution does not affect its notifier function, if any; the notifier function remains installed.

SEE ALSO

Provider events are described in "Provider Events" on page 2-7.

Modes of operation and notifier functions are described in "Specifying How Provider Functions Execute" on page 2-9.

To set a provider to asynchronous mode, call the `OTSetAsynchronous` function (page 2-30). To find out a provider's mode of execution, call the `OTIsSynchronous` function (page 2-31).

OTIsSynchronous

Returns a provider's current mode of execution.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference for the provider whose mode of execution you want to obtain.

DESCRIPTION

The `OTIsSynchronous` function returns `true` if a provider is in synchronous mode or returns `false` if the provider is in asynchronous mode.

SEE ALSO

To set a provider to synchronous mode, call the `OTSetSynchronous` function (page 2-29). To set a provider to asynchronous mode, call the `OTSetAsynchronous` function (page 2-30).

OTCancelSynchronousCalls

Cancels any currently executing synchronous function for a specified provider.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference for the provider whose synchronous function you want to cancel.

DESCRIPTION

The `OTCancelSynchronousCalls` function cancels any currently executing synchronous function for the provider that you specify. The provider need not be in synchronous mode when you call this function.

Typically, you would call the `OTCancelSynchronousCalls` function at interrupt time by installing a Time Manager task that executes after a given amount of

time has passed. You could do this to prevent a synchronous function from hanging the system.

IMPORTANT

The `OTCancelSynchronousCalls` function 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. For example, calling the `OTCancelSynchronousCalls` function on a connection-oriented endpoint might break its connection and render the endpoint unusable. ▲

SEE ALSO

To set a provider to synchronous mode, call the `OTSetSynchronous` function (page 2-29). To find out a provider's current mode of execution, call the `OTIsSynchronous` function (page 2-31).

Time Manager tasks are described in the Time Manager chapter of *Inside Macintosh: Processes*.

OTSetBlocking

Allows a provider to wait or block until it is able to send or receive data.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider that is to block.

DESCRIPTION

The `OTSetBlocking` function causes provider functions that send or receive data to wait if current conditions prevent them from completing an operation. By default, a provider is in nonblocking mode, in which case, if a provider function were unable to complete sending or receiving data, it would return immediately with a result that would tell you why the operation was unable to complete.

If a provider is in blocking mode and you call the `OTCloseProvider` function to close the provider, Open Transport gives each Streams module up to 15 seconds to process outgoing commands. It is recommended that you call the `OTSetNonBlocking` function before you call the `OTCloseProvider` function.

SEE ALSO

Blocking is described in "Setting a Provider's Blocking Status" on page 2-10.

To set a provider's blocking status to nonblocking, call the `OTSetNonBlocking` function (page 2-34). To find out a provider's blocking status, call the `OTIsNonBlocking` function (page 2-35).

Blocking attributes affect endpoint providers more than other providers. For more information see the discussion about modes of operation in the chapter "Endpoints" in this book.

OTSetNonBlocking

Does not allow a provider to wait if it cannot currently complete a function that sends or receives data.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider whose blocking mode is being set.

DESCRIPTION

The `OTSetNonBlocking` function causes provider functions to return a result code immediately, instead of waiting for a function that sends or receives data to complete. When you open a provider, its mode of operation is set to nonblocking by default.

If a provider is in nonblocking mode and you call the `OTCloseProvider` function, the provider flushes all outgoing commands in the stream and immediately close the provider. Conversely, in blocking mode, the provider would give each Streams module up to 15 seconds to flush outgoing commands. It is recommended that you call the `OTSetNonblocking` function before you call the `OTCloseProvider` function.

SEE ALSO

Blocking is described in "Setting a Provider's Blocking Status" on page 2-10.

To set a provider's blocking status to blocking, call the `OTSetBlocking` function, (page 2-33). To find out a provider's blocking status, call the `OTIsNonBlocking` function (page 2-35).

Blocking attributes affect endpoint providers more than other providers. For more information, see the discussion about modes of operation in the chapter "Endpoints" in this book.

OTIsNonBlocking

Returns a provider's current blocking status.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider whose blocking status is sought.

DESCRIPTION

The `OTIsNonBlocking` function returns `true` if the provider's current blocking status is nonblocking or returns `false` if it is blocking.

SEE ALSO

Blocking is described in "Setting a Provider's Blocking Status" on page 2-10.

To set a provider's blocking status to blocking, call the `OTSetBlocking` function (page 2-33). To set a provider's blocking status to nonblocking, call the `OTSetNonBlocking` function (page 2-34).

Blocking attributes affect endpoint providers more than other providers. For more information see the discussion about modes of operation in the chapter "Endpoints" in this book.

OTAckSends

Specifies that a provider make an internal copy of data being sent and that it notify you when it has finished sending data.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider that is sending data.

DESCRIPTION

By default, providers make an internal copy of data before sending it and they do not acknowledge sends. If you use the `OTAckSends` function to specify that the provider acknowledge sends and you call a function that sends data, the provider does not copy the data before sending it. Instead, it reads data directly from your buffer while sending. For this reason, you must not change the contents of your buffer until the provider is no longer using it. The provider lets you know that it has finished using the buffer by calling your notifier function and passing `T_MEMORYRELEASED` event code for the `code` parameter, a pointer to the buffer that was sent in the `cookie` parameter, and the size of the buffer in the `result` parameter.

If you have not installed a notifier function for the provider, this function returns the `kOTAccessErr` result.

If a send is currently outstanding on the provider, from a call to the `OTSnd`, `OTSndUData`, `OTSndUReply`, `OTSndURequest`, `OTSndReply`, or `OTSndrequest` function, the `OTAckSends` function returns a `kOTChangeStateErr` message.

▲ **WARNING**
You need to be sure that are no outstanding `T_MEMORY_RELEASED` events for a provider before you close the provider. Otherwise, Open Transport attempts to deliver the event to a provider that no longer exists, with unpredictable results, such as crashing the system. ▲

SPECIAL CONSIDERATIONS

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.

SEE ALSO

To request that the provider copy the data before sending it, use the `OTDontAckSends` function, described in the next section.

To find out a provider's current send-acknowledgment status, call the `OTIsAckingSends` function (page 2-39).

For additional information, see "Setting a Provider's Send-Acknowledgment Status" on page 2-10.

The send-acknowledgment status of a provider is ignored by mapper providers, AppleTalk providers, and TCP/IP providers. For information about how endpoint providers are affected, see the discussion of an endpoint's mode of operation in the chapter "Endpoints" in this book.

OTDontAckSends

Specifies that a provider copy data before sending it.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider that is sending data.

DESCRIPTION

By default, providers do not acknowledge sends. You need to call the `OTDontAckSends` function only if you have used the `OTAckSends` function to turn on send-acknowledgment for a provider.

If a send is currently outstanding on the provider, from a call to the `OTSnd`, `OTSndUData`, `OTSndUReply`, `OTSndURequest`, `OTSndReply`, or `OTSndrequest` function, the `OTDontAckSends` function returns a `kOTChangeStateErr` message.

SEE ALSO

To prevent buffer copying and request completion events for provider functions that send data, call the `OTAckSends` function, described on page 2-36.

To find out whether a provider is acknowledging sends, call the `OTIsAckingSends` function (page 2-39).

For additional information, see "Setting a Provider's Send-Acknowledgment Status" on page 2-10.

OTIsAckingSends

Determines whether a provider is acknowledging sends.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider sending data.

DESCRIPTION

The `OTIsAckingSends` function returns `true` if the provider acknowledges sends and `false` if it does not.

SEE ALSO

To specify that a provider acknowledge sends, call the `OTAckSends` function (page 2-36). To specify that a provider not acknowledge sends, call the `OTDontAckSends` function (page 2-38).

For additional information, see "Setting a Provider's Send-Acknowledgment Status" on page 2-10.

Installing and Removing a Notifier Function

To receive notice of provider events, you must install a notifier function. If the provider is synchronous, you do this by calling the `OTInstallNotifier` function. If the provider is asynchronous, you install the notifier by passing a pointer to the notifier function as a parameter to the function used to open the provider. To remove a notifier function, call the `OTRemoveNotifier` function.

OTInstallNotifier

Installs a notifier function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider for which you are installing a notifier.
`proc`	A pointer to a notifier function that you provide. For C++ applications, the `proc` parameter must point to either a C function or a static member function.
`contextPtr`	A context pointer for your use. The provider passes this value unchanged to your notifier function when it calls the function.

DESCRIPTION

The `OTInstallNotifier` function installs a notifier function for the provider that you specify. Changing a provider's mode of execution does not affect the notifier function. The notifier function remains installed until you remove it using the `OTRemoveNotifier` function or until you close the provider.

Before calling the `OTInstallNotifier` function, you must open the provider for which you want to install the notifier. If you open a provider asynchronously (for example, with the `OTAsyncOpenEndpoint` function), you must pass a pointer to a notifier function as a parameter to the function used to open the provider. In this case, you don't need to call the `OTInstallNotifier` function unless you want to install a different notifier function. If you do, you must call the `OTRemoveNotifier` function before calling the `OTInstallNotifier` function.

Opening a provider synchronously (for example, with the `OTOpenEndpoint` function) opens the provider but does not install a notifier function for it. If you need a notifier function for a provider opened synchronously, you must call the `OTInstallNotifier` function. This notifier would not return completion events, but would return asynchronous events advising you of the arrival of data, of changes in flow-control restrictions, and so on.

Call the `OTInstallNotifier` function only when no provider functions are executing for the provider that you specify. Otherwise, the `OTInstallNotifier` function returns the result code `kOTStateChangeErr`.

If you try to install a notifier function for a provider that already has a notifier, the function returns with the `kOTAccessErr` result.

IMPORTANT

On 68000-based Macintosh computers, the `OTInstallNotifier` function saves the current value of the A5 register. Open Transport restores the A5 register to this saved value when calling the notifier function you install. If your environment stores context information in a register other than A5, your notifier function must save and restore the value of that register. ▲

SEE ALSO

Notifier functions are described in "Application-Defined Functions" (page 2-45).

To remove an installed notifier function, call the `OTRemoveNotifier` function, described in the next section.

OTRemoveNotifier

Removes a provider's notifier function.

C INTERFACE

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

C++ INTERFACE

```
OSStatus TProvider::RemoveNotifier();
```

PARAMETERS

`ref`	A provider reference for the provider whose notifier function is to be removed.

DESCRIPTION

The `OTRemoveNotifier` function removes the notifier (if any) currently installed for the provider specified by the `ref` parameter.

SEE ALSO

Notifier functions are described in the section "Application-Defined Functions" (page 2-45).

To install a notifier function, call the `OTInstallNotifier` function (page 2-40).

Sending Module-Specific Commands

You can define module-specific commands for an Open Transport protocol module. Open Transport does not interpret these commands; it merely relays them from your application to the protocol module. You can send a module-specific command to an Open Transport protocol module by using the `OTIoctl` function.

OTIoctl

Sends a module-specific command to an Open Transport protocol module.

C INTERFACE

```
SInt32 OTIoctl(ProviderRef ref, UInt32 cmd, void* data);
```

C++ INTERFACE

```
SInt32 TProvider::Ioctl(UInt32 cmd, void* data);
```

PARAMETERS

`ref`	The provider reference of the provider affected by the specified command.
`cmd`	A routine selector for the module-specific command.
`data`	Data to be used by the module-specific command, or a pointer to such data. The interpretation of the `data` parameter is command specific.

DESCRIPTION

The `OTIoctl` function sends a module-specific command to an Open Transport protocol module. The `OTIoctl` function runs synchronously or asynchronously, matching the provider's mode of execution.

If the `OTIoctl` function completes synchronously without error, it returns 0 or a positive integer. The positive integer's meaning is command specific. If the `OTIoctl` function fails while executing synchronously, its return value is a negative integer corresponding to an Open Transport result code.

If the `OTIoctl` function runs asynchronously, it returns immediately with a return value `kOTNoError` or another Open Transport result code. When the function completes execution, Open Transport calls the notifier function you specify, passing the event code `kStreamIoctlEvent` and a `result` parameter indicating the result of the completed `OTIoctl` function. If the value of the `result` parameter is greater than 0, the corresponding result code is defined by the command; otherwise, the value of the `result` parameter corresponds to an Open Transport result code.

SPECIAL CONSIDERATIONS

Using the `OTIoctl` function makes your application module dependent; you should not use the `OTIoctl` function if you want your application to be transport independent.

SEE ALSO

Positive return values for the `OTIoctl` function are defined by the Open Transport module that you are using. Refer to the documentation for that module for information.

Application-Defined Functions

To receive notice of provider events, you must write and install a notifier function. A notifier function is the callback function that a provider uses to communicate information back to your application for all events affecting a particular provider. A provider in asynchronous mode must have a notifier function to receive completion events.

Most providers must also use a notifier function to retrieve asynchronous events. An endpoint provider can poll for asynchronous events using the `OTLook` function, but a mapper provider or a service provider cannot poll for asynchronous events; it must use a notifier function instead. In general, it is recommended that all providers use notifier functions to handle both asynchronous and completion events.

MyNotifierCallbackFunction

After you install a notifier function on a provider, the provider calls the notifier function each time an Open Transport event occurs for that provider.

C INTERFACE

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

C++ INTERFACE

In C++, a notifier function can be either a C function (as in the C interface) or a member function of a class—the `TOTProcNotifier` class, the `TOTMethodNotifier` class, or a subclass of these. The return type and parameters of a notifier function are the same whether the notifier function is a C function or a C++ member function.

PARAMETERS

`contextPtr`	The value you specified for the `contextPtr` parameter when installing this notifier function. You can use this parameter in any way that is useful to you. If you do not need it, you can set the pointer to `nil`.
`code`	An event code indicating the event that occurred. Possible values for event codes are given in the event code enumeration (page 2-17).
`result`	For completion events, the result code of the completed provider function, identified by the `code` parameter. For completion events, the meaning of the `result` parameter is event specific. (For most asynchronous events, the `result` parameter has no meaning and can be ignored.) For additional information, see the description of the individual function.
`cookie`	A pointer to data. The meaning and type of the data vary depending on the function that has completed executing. For additional information, see the event codes enumeration (page 2-17).

DESCRIPTION

Using a notifier function is the recommended way for your application to handle completion and asynchronous events. After you install a notifier function for a provider, the function is called by the provider each time an Open Transport event occurs for that provider. For a completion event, the provider passes the function result in the `result` parameter, the event code in the `code` parameter, and any additional information in the `cookie` parameter. For an asynchronous event, the provider usually passes the event code in the `code` parameter and passes no other information.

Open Transport calls your notifier function at secondary interrupt level (deferred task time). For this reason, your notifier function is subject to the same rules and restrictions as are all Macintosh functions that can be called at interrupt time; these restrictions are summarized in the section "Using Notifier Functions to Handle Provider Events," beginning on page 2-13.

You can install the same notifier function for two or more providers. But each time you install the same notifier function for a different provider, you must pass a different value in the `contextPtr` parameter of the function that installs the notifier. The data structure referenced by the `contextPtr` parameter points

must include a provider reference or some other identifier that uniquely identifies the provider for which the notifier is called.

SPECIAL CONSIDERATIONS

The following information applies to applications written for 68000-family machines. Before calling your notifier function, Open Transport restores the A5 register to the value it had when you installed the notifier function. Thus, if your development environment saves your application context in the A5 register, your notifier function need not restore its A5 world. But if your development environment saves your application context in a register other than A5, your notifier function must save and restore that register.

SEE ALSO

To install a notifier function for an existing provider, call the `OTInstallNotifier` function (page 2-40). You can also install a provider when you open a provider asynchronously by passing a pointer to the notifier function as a parameter to the function used to open the provider. For additional information, see the reference section of the chapter describing the provider of interest.

To remove a notifier function, call the `OTRemoveNotifier` function (page 2-42).

For a list and description of event codes see the event codes enumeration (page 2-17).

For an example of a notifier function, see Listing 2-1 on page 2-14.

Endpoints

Contents

This chapter explains how your application can use endpoints to communicate with endpoint providers, the layered set of protocol modules that provide data transfer services. The chapter describes

- the services offered by different types of endpoint providers
- the concept of transport independence and the use of options
- how an endpoint's mode of execution and mode of operation affect the behavior of endpoint functions
- how you use endpoint functions to obtain information about endpoints, to establish connections, and to transfer data

To understand this chapter, you must first read the chapters "Introduction to Open Transport" and "Providers," which introduce many of the concepts discussed and further elaborated in this chapter.

This chapter offers minimal information about options, values you can specify that link the behavior of your application to the specific configuration of an endpoint provider. For information about options, you must read the chapter "Option Management" in this book.

About Endpoints

An **endpoint** is the communications path between your application and an **endpoint provider,** which is a layered set of protocols that define how data and other information are exchanged between you and a remote client. The endpoint consists of a set of data structures, maintained by Open Transport, that specify the components of the endpoint provider, and the manner in which the provider is to operate. In the process of opening an endpoint, you configure the endpoint provider and specify the protocol or set of protocols you want to use to transfer data and, if required, the hardware link. The chapter "Configuration Management" in this book explains how you specify the software and hardware support your application requires. Whether you specify a single protocol or a layered set of protocols, the type of service provided by the highest-level protocol defines the type of the endpoint. For example, if you specify the AppleTalk Transaction Protocol (ATP), which offers connectionless transaction-based service, the endpoint is a connectionless transaction-based endpoint.

When you open an endpoint, Open Transport creates a data structure that contains information about the services the endpoint provider offers, the limits on the size of data it can send and receive, the size of internal buffers used to hold data, the current state of the endpoint, and so on. Open Transport obtains this information from the particular protocol implementations that you specify when you configure the endpoint provider. You can access information in some fields of this structure by calling functions that return information about the endpoint provider. Other fields of the structure are private and can be accessed only by Open Transport.

Opening an endpoint also creates an **endpoint reference,** an number that uniquely identifies this endpoint and that you must specify when calling any function relating to this endpoint.

Before you can use the endpoint to transfer data, you must **bind** the endpoint—that is, you must associate the endpoint with a logical address. Depending on the protocol you use, you can specify this address as a symbolic name or as a network address. Specific address binding rules and address formats also vary with the protocol you use. In general, you cannot bind more than one connectionless endpoint to an address, but you can bind several connection-oriented endpoints to a single address.

Open Transport functions that you can use only with endpoints are called **endpoint functions.** You use endpoint functions to create and bind an endpoint, to obtain information about an endpoint, to establish and break down connections, and to transfer data. What functions you can call for an endpoint depends on the type of the endpoint and on its state. The behavior of a function is determined by the endpoint's mode of operation. In order to write Open Transport applications that behave in a reliable and predictable manner, it is important that you understand how these factors affect the behavior of an endpoint provider. The rest of this section describes the different types of endpoints, describes the effect of an endpoint's mode of operation on the behavior of endpoint functions, and explains how Open Transport uses information about endpoint states to manage endpoints.

Endpoint Types and Mode of Service

There are four types of endpoints, each offering a different *mode of service*:

- connection-oriented transactionless service

  Either endpoint can initiate this type of service. It allows for the transfer of very large amounts of data with guaranteed data delivery and a reliable data stream.

- connection-oriented transaction-based service

  Either endpoint can initiate the connection, but only the endpoint sending the request can initiate a transaction. Using this service, you can conclude an unlimited number of parallel transactions. This service guarantees delivery and can detect duplicate sends.

- connectionless transactionless service

  Either endpoint can initiate this type of service. Some protocols can calculate checksums for incoming packets, but generally this service provides only best-effort delivery and allows the transfer of relatively small amounts of data at one time.

- connectionless transaction-based service

  Only the endpoint sending a request can initiate this type of service. It allows for the transfer of larger amounts of data, provides some error checking, detects duplicate sends, and guarantees that response packets are delivered in the order sent.

As you can tell from the foregoing description, in Open Transport there is no such thing as a connectionless endpoint. It would have to be either a connectionless transaction-based endpoint or a connectionless transactionless endpoint. However, because there are issues that affect endpoints inasmuch as they are connectionless and not connection-oriented or transactionless and not transaction-based, when this chapter identifies an endpoint using only one service name, you should assume that the endpoint can be in one of two modes of service. Thus, the term *transaction-based endpoint* can refer either to a connectionless transaction-based endpoint or to a connection-oriented transaction-based endpoint.

The chapter "Introduction to Open Transport" in this book defines and describes the different services that each type of endpoint offers and explains some of the criteria you might use for selecting a specific type. The documentation for the protocol you are using provides information about how a mode of service is implemented for your endpoint and the options that you can use to fine-tune its behavior. The section "Using Endpoints" beginning on page 3-20 describes how you use endpoint functions to implement these services. However, before you read that section, you might find it helpful to understand the naming conventions used for endpoint functions because these are directly related to an endpoint's mode of service. These conventions are described in the next section.

Naming Conventions for Endpoint Functions

You can use endpoint functions that return information about the endpoint's state, address, mode of execution, or mode of operation with all endpoint types. However, which endpoint functions you can call to transfer data depends on the type of the endpoint. There is no single function that you can use to send data or to receive data. For each type of endpoint you open, you must use a send function that is specific to that type. For example, when you send data using a connectionless transactionless endpoint, you call the `OTSndUData` function; when you send data using a connection-oriented transactionless endpoint, you call the `OTSnd` function. Table 3-1 presents a summary of the function names for functions used to transfer data. The functions are grouped together based on the endpoint's mode of service. Look over this table briefly and see if you can spot the distinguishing trait for each group of names.

Table 3-1 The names of functions used to transfer data

	Connectionless	Connection-oriented
Transactionless	`OTSndUData`	`OTSnd`
	`OTRcvUData`	`OTRcv`
	`OTRcvUDErr`	
Transaction-based	`OTSndURequest`	`OTSndRequest`
	`OTRcvURequest`	`OTRcvRequest`
	`OTSndUReply`	`OTSndReply`
	`OTRcvUReply`	`OTRcvReply`
	`OTCancelURequest`	`OTCancelRequest`
	`OTCancelUReply`	`OTCancelReply`

The following bulleted items explain the rationale for the conventions used to name the different groups of functions:

- Transaction-based endpoints send and receive requests and replies. If a function name contains the string “Request” or “Reply,” you use the function for a transaction-based endpoint; for example, `OTSndURequest` or `OTSndRequest`.
- Transactionless endpoints send and receive data, not requests or replies. If a function name contains the string “Snd” or “Rcv” but does not contain “Request” or “Reply,” you use the function for a transactionless endpoint; for example, `OTSnd` or `OTSndUData`.
- Connectionless endpoints must include the destination address as a parameter to every send operation and the source address as a parameter to every receive operation. This is signalled by the letter “U” in the name of a function. Thus, you call the `OTSndUData` function for a connectionless transactionless endpoint, but you call the `OTRcvURequest` function for a connectionless transaction-based endpoint.

- Connection-oriented endpoints do not need to include addresses in send and receive operations because establishing the connection also determines the addresses, which do not change during a session. The names of functions that can be called for connection-oriented endpoints are exactly the same as for connectionless endpoints except that the "U" is omitted. Thus, you call the `OTSnd` function for a connection-oriented transactionless endpoint and the `OTSndRequest` function for a connection-oriented transaction-based endpoint.

Of course, you can use the functions that establish and tear down connections only with connection-oriented endpoints. These functions suggest their use in their names: for example, `OTConnect` or `OTSndDisconnect`. Connection-oriented endpoints support two kinds of disconnects: abortive disconnects and orderly disconnects. An **abortive disconnect** breaks a connection immediately, even if this were to result in loss of data; an **orderly disconnect** allows an endpoint to send all data remaining in its send buffer before it breaks a connection. These two kinds of disconnects are reflected in the names of the functions used: `OTSndDisconnect` for an abortive disconnect and `OTSndOrderlyDisconnect` for an orderly disconnect.

Endpoint Options

The goal of Open Transport is to allow one type of endpoint to communicate with the same type of endpoint (or with a remote client offering the same mode of service) simply by having the application reconfigure the endpoint provider so as to use the protocol of the remote client. Reconfiguring the endpoint provider would require very minimal changes to the application and consequently make your application virtually independent of the underlying transport used to transfer data. Achieving transport independence, however, also means being willing to forego any special advantages or features that a protocol has to offer. If it is not possible for you to do without these features, you can use options to take advantage of protocol-specific features. An **option** is a value that you can set for an endpoint, which links the behavior of your application to the specific protocol that you have used to configure the endpoint provider. By using options, you can take advantage of a service that is unique to a protocol.

In general, the use of options decreases the portability of your application. When you open an endpoint, the endpoint provider creates a buffer containing default option values that it chooses to ensure maximum portability across protocol families and system platforms. It is recommended that you use these values rather than setting different values. However, if you need to customize transport services, you might need to specify different option values. Selecting alternate option values begins a process called **option negotiation.** During this process, option values are negotiated between an endpoint and its provider or, if the option affects a connection or transaction, between a local and remote endpoint and their providers. The providers must conclude this negotiation process before you can use an endpoint to transfer data. Besides noting those instances in which you can specify option information when calling endpoint functions, this chapter provides no information about options. For detailed information about options and for a description of the `OTOptionManagement` endpoint function, see the chapter "Option Management" in this book.

Modes of Operation

An endpoint provider, like other Open Transport providers, can also be characterized by its mode of operation, which determines

- whether the functions used for that endpoint provider execute synchronously or asynchronously.

  The chapter "Providers" in this book contains a detailed discussion of the issues involved in selecting one or another mode of execution. The section "Handling Events for Endpoints," beginning on page 3-24 offers additional information about how an endpoint provider's mode of execution specifically affects endpoint functions.

- whether the provider blocks or waits when sending or receiving data and
- whether the provider copies data that you want to send before sending it.

The chapter "Providers" also introduces these concepts and describes the functions you use to get and set a provider's mode of operation. The rest of this section contains a more detailed discussion of how blocking and acknowledging sends specifically affect endpoint functions.

Blocking

If an endpoint provider is blocking, functions that you use to send or receive data do not complete until they actually write or read the amount of data that you have specified should be written or read.

- You specify the amount of data you expect to write, by setting the `len` field of a `TNetBuf` structure to the length of the data in the data buffer. If the size of data in the data buffer is smaller than the size you specified in the `len` field, the function will not complete. Under the same circumstances, if the endpoint is not blocking, the function will complete.
- You specify the amount of data you expect to read, by setting the `maxlen` field of a `TNetBuf` structure. If the size of the incoming data is smaller than the value specified in the `maxlen` field, the function will not complete. Under the same circumstances, if the endpoint is not blocking, the function will complete.

If you are sending data faster than the network can handle it, this gives rise to flow-control restrictions. If an endpoint is blocking, a send function waits until flow-control restrictions are lifted before it executes. A send function must also wait if an endpoint provider cannot deal with a request immediately, but must queue the request before it is able to handle it.

If an endpoint provider is nonblocking or asynchronous and a send function cannot complete due to flow-control restrictions, the function returns with the `kOTFlowErr` result or it returns a positive integer. If the function returns the `kOTFlowErr` result, this means that it has not been able to send any data; if it returns a positive integer, this represents the amount of data it has been able to send. When flow-control restrictions are lifted, the provider issues a `T_GODATA` or `T_GOEXDATA` event. Upon receiving this event, you should execute the send function again to send the remaining data.

If an endpoint provider is nonblocking or asynchronous and a send function cannot complete because the request for function execution would have to be queued, the function returns with the `kEAGAINErr` or `kEWOULDBLOCKErr` result. You should try to execute the command later.

If an endpoint provider is in synchronous blocking mode and a receive function cannot complete because the data has not arrived, the function does not return until either data actually arrives and the size of the data is equal to the maximum size you specified for the receive buffer, or data arrives and the

T_MORE flag is not set (there's an EOM marker, which means that you have retrieved all the data sent). If an endpoint provider is nonblocking and a receive function cannot complete because data has not yet arrived, the function returns with the kOTNoDataErr. You should try calling the function again later.

An endpoint provider is nonblocking unless you use the OTSetBlocking function to change its mode of operation.

Acknowledging Sends

You can also affect the behavior of functions that send data by specifying that the endpoint provider acknowledge sends. By default, Open Transport does not acknowledge the completion of send operations. This means that when you call a function to send data, Open Transport copies the data from the client buffer into a different buffer and then sends it. If you ask Open Transport to acknowledge sends, it relies on the fact that your data buffer will remain stable until the endpoint provider can actually send the data. After it sends the data, the provider calls your notifier function passing T_MEMORYRELEASED for the code parameter, a pointer to the buffer that was sent in the cookie parameter, and the size of the buffer in the result parameter.

Endpoint States

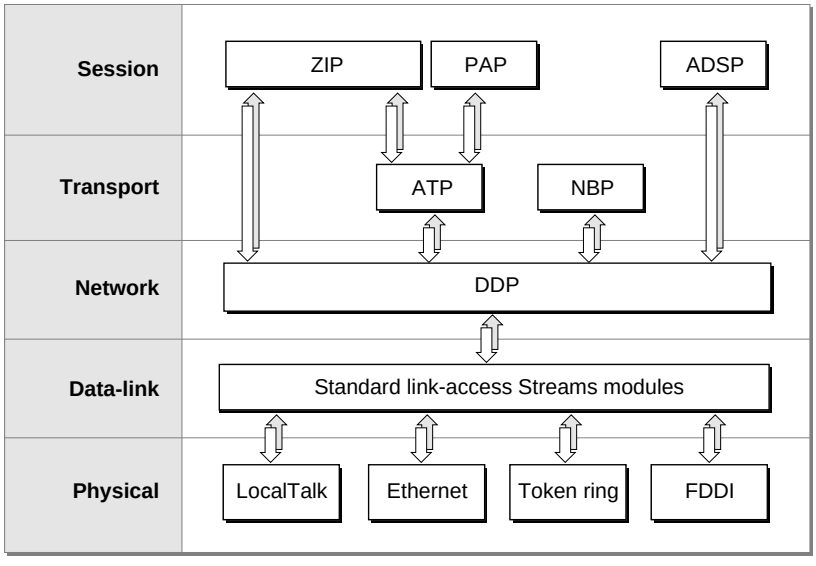
Each endpoint has an attribute known as its **endpoint state.** An endpoint state governs which endpoint functions you can call for the endpoint. For example, if you open an endpoint but do not bind it, it is in the T_UNBND state and the only two functions you can call for the endpoint are OTCloseProvider or OTBind. The endpoint's mode of service determines the possible states an endpoint can be in while it is transferring data. For example, a connectionless endpoint can only transfer data while it is in the T_IDLE state; a connection-oriented endpoint can only transfer data while it is in the T_DATAXFER state. Table 3-2 describes possible endpoint states for connectionless and connection-oriented endpoints and suggests in parentheses an English equivalent for the name of each constant.

Table 3-2 Endpoint states

State	Meaning
T_UNINIT	This endpoint has been closed and destroyed or has not been used. (Uninitialized)
T_UNBND	This endpoint is initialized but has not yet been bound to an address. (Unbound)
T_IDLE	This endpoint has been bound to an address and is ready for use. Connectionless endpoints can send or receive data; connection-oriented endpoints can initiate or listen for a connection. (Idle)
T_OUTCON	This connection-oriented endpoint has initiated a connection and is waiting for the peer endpoint to accept the connection. (Outgoing connection request)
T_INCON	This connection-oriented endpoint has received a connection request but has not yet accepted or rejected the request. (Incoming connection request)
T_DATAXFER	This connection-oriented endpoint can now transfer data because the connection has been established. (Data transfer mode)
T_OUTREL	This connection-oriented endpoint has issued an orderly disconnect that the peer endpoint has not acknowledged. The endpoint can continue to read data but must not send any more data. (Outgoing release request)
T_INREL	This connection-oriented endpoint has received a request for an orderly disconnect, which it has not yet acknowledged. The endpoint can continue to send data until it acknowledges the disconnection request, but it must not read data. (Incoming release request)

Figure 3-1 shows a diagram illustrating the possible endpoint states for a connectionless endpoint.

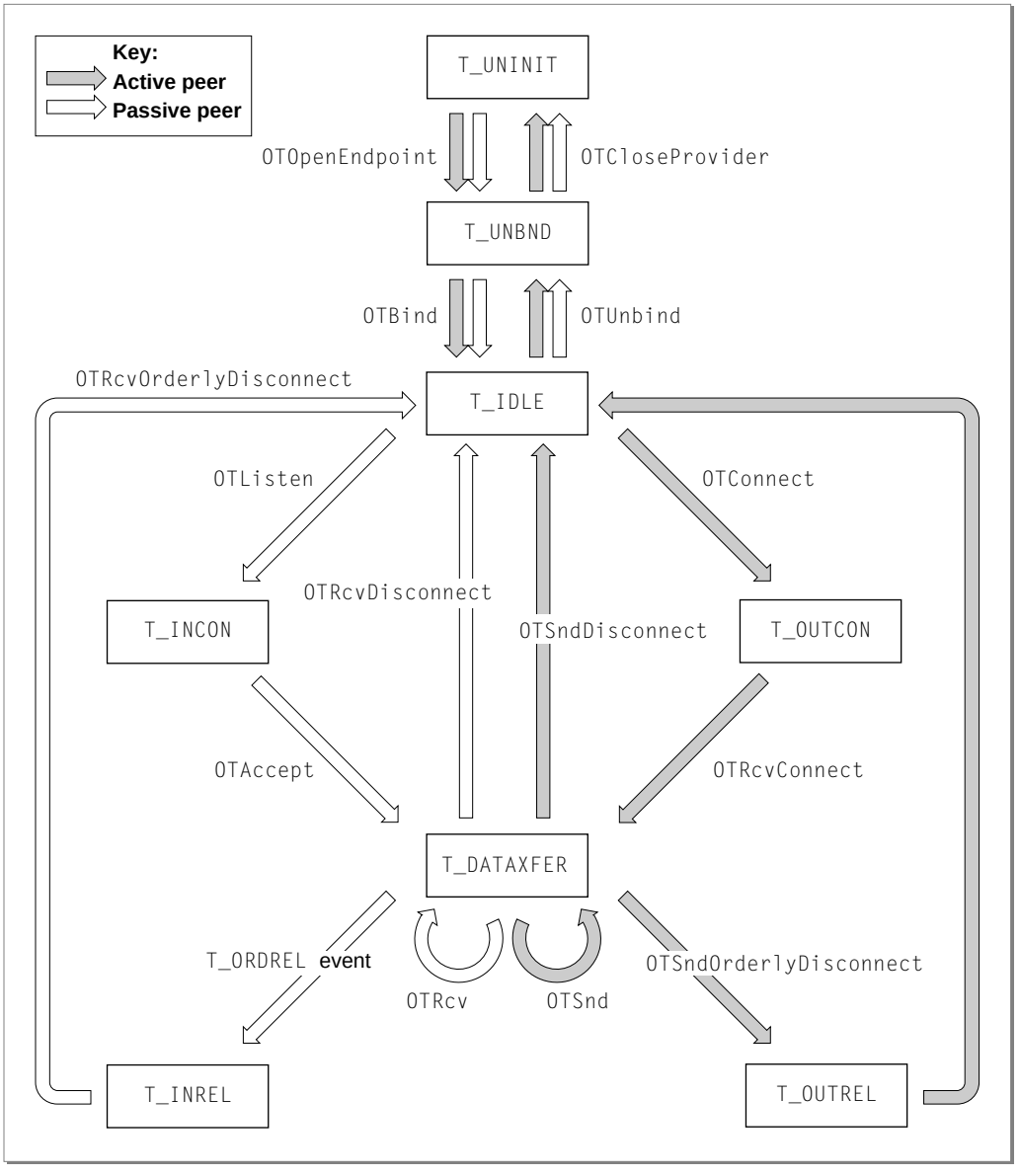
Figure 3-1 Possible endpoint states for a connectionless endpoint



A connectionless endpoint can be in one of three states: `T_UNINIT`, `T_UNBND`, or `T_IDLE`. Before you open the endpoint, it is in the `T_UNINIT` state. After you open the endpoint but before you bind it, it is in the `T_UNBND` state. After you bind the endpoint, it is in the `T_IDLE` state and is ready to transfer data. A connectionless transactionless endpoint would use the `OTSndUData` or `OTRcvUData` functions to transfer data; a connectionless transaction-based endpoint would use the `OTSndURequest`, `OTRcvURequest`, `OTSndUReply`, and `OTRcvUReply` functions to transfer data. When the endpoint finishes transferring data, you must first unbind the endpoint—that is, dissociate the endpoint from its address. At this stage, the endpoint returns to the `T_UNBND` state. Then you can close the endpoint, at which time the endpoint returns to the `T_UNINIT` state.

Figure 3-2 shows a state diagram illustrating the possible endpoint states for a connection-oriented endpoint.

Figure 3-2 Possible endpoint states for a connection-oriented endpoint



Like a connectionless endpoint, a connection-oriented endpoint is in the `T_UNINIT` state until you open it and then, in the `T_UNBND` state until you bind it. After you bind an endpoint but before you inititate a connection, an endpoint is in the `T_IDLE` state.

During the connection process, the endpoint provider initiating the connection, known as the **active peer,** calls the `OTConnect` function to request a connection. At this point, the active peer is in the `T_OUTCON` state. The endpoint provider listening for a connection request, known as the **passive peer,** calls the `OTListen` function to read an incoming request. After it has read the request, the passive peer changes to the `T_INCON` state. It can now either accept the connection using the `OTAccept` function or reject the connection using the `OTSndDisconnect` function. If the endpoint accepts the connection, it changes to the `T_DATAXFER` state; if it rejects the connection it goes back to the `T_IDLE` state.

The active peer must acknowledge the response using the `OTRcvConnect` function (for a connection that has been accepted) or the `OTRcvDisconnect` function (for a connection that has been rejected). Calling the `OTRcvConnect` function establishes the connection and places the active peer in the `T_DATAXFER` state. Calling the `OTSndDisconnect` function rejects the connection and places the active peer in the `T_IDLE` state. After they are connected, endpoints can transfer data using simple send and receive operations or using transaction requests and replies, depending on whether the endpoint is transactionless or transaction-based.

When the client applications have finished transferring data, they must tear down the connection by using an orderly disconnect process if possible. That is, the active peer, should check to see whether the protocol supports an orderly disconnect. If it does, the active peer initiates this process by calling the `OTSndOrderlyDisconnect` function. This places the active peer in the `T_OUTREL` state. It also creates a pending `T_ORDREL` event for the other endpoint. The passive peer can retrieve the event using a notifier function or using the `OTLook` function. It must then acknowledge receiving the disconnection request by calling the `OTRcvOrderlyDisconnect` function. Then it must tear down its side of the connection by also calling the `OTSndOrderlyDisconnect` function, which the other side must also acknowledge. Disconnecting the endpoints places them in the `T_IDLE` state again, and you can reconnect or close them.

Open Transport uses endpoint state information to manage endpoints. Consequently, it is crucial that you call functions in the right sequence and that you call functions to acknowledge receipt of data as well as of connection and disconnection requests. Sending these acknowledgments is necessary to leave the endpoint in an appropriate state for further processing. In your application,

you can sometimes use the `OTGetEndpointState` function to determine an endpoint's state, which is one more way to test for successful completion of a function.

Table 3-3 lists the functions that can change an endpoint's state and specifies what the resulting state is depending on whether the function succeeds or fails.

Table 3-3 Functions that can change an endpoint's state

State before call	Function	State after call: No error	State after call: If error
`T_UNINIT`	`OTOpenEndpoint`	`T_UNBND`	N/A
Any	`CloseProvider`	`T_UNINIT`	N/A
`T_UNBND`	`OTBind`	`T_IDLE`	`T_UNBND`
`T_IDLE`	`OTUnbind`	`T_UNBND`	N/A
`T_IDLE`	`OTConnect`	`T_OUTCON`	`T_IDLE`
`T_OUTCON`	`OTRcvConnect`	`T_DATAXFER`	`T_IDLE`
`T_INCON`	`OTAccept`	`T_DATAXFER`	`T_IDLE`

`T_DATAXFER`	`OTSndDisconnect`
	`OTSndOrderlyDisconnect`
	`OTRcvDisconnect`
	`OTRcvOrderlyDisconnect`
		`T_IDLE`	`T_DATAXFER`

The arrival of an asynchronous event can also change the state of an endpoint. Table 3-4 shows the state of the endpoint before the event is received and the state of the endpoint after the event is consumed. An event is consumed or cleared when your application acknowledges receipt of the event. For example, if you get a `T_LISTEN` event, you call the `OTListen` function; after you get a `T_DISCONNECT` event, you call the `OTRcvDisconnect` function.

Table 3-4 Events that can change an endpoint's state

Old State	Event	New State
T_IDLE	T_LISTEN	T_INCON
T_IDLE	T_CONNECT	T_DATAXFER
T_IDLE	T_PASSCON	T_DATAXFER
T_OUTCON, T_DATAXFER	T_DISCONNECT	T_IDLE
T_DATAXFER	T_ORDREL	T_INREL

The section "Handling Events for Endpoints" on page 3-24 lists the asynchronous events that a provider can issue and the functions you must call to clear these events.

Transport Service Data Units

The main purpose of endpoints is to transfer data. The terms *transport service data unit* and *expedited transport service unit* are used to describe the size and kind of data that a particular endpoint can handle when it is transferring data in discrete units known as *datagrams*. Not all protocols use transport service data units to transfer data.

A **transport service data unit** (**TSDU**), whether it is normal or expedited, refers to the largest piece of data that an endpoint can transfer with boundaries and content preserved unchanged. Different types of endpoints and different endpoint implementations support different size TSDUs.

An **expedited transport service data unit** (**ETSDU**), refers to the largest piece of expedited data than an endpoint can transfer. **Expedited data** is considered to be urgent. An endpoint provider that can handle expedited data guarantees that this data takes precedence over any other normal data that is being transmitted. Not all endpoint providers can transfer expedited data. Usually, connection-oriented and transaction-based endpoints require the use of expedited data for control or attention messages, and therefore the implementation of these types of endpoints often supports the transfer of expedited data.

When you open an endpoint, Open Transport creates an endpoint information structure, a `TEndpointInfo` structure, that you can examine to find out whether the endpoint supports normal or expedited data and the maximum size of this data. The section "Obtaining Information About Endpoints," beginning on page 3-23 explains how you examine this structure to find out this information.

Using Endpoints

This section begins by explaining how you create an endpoint and associate it with an address. Next, it introduces the functions you can use to obtain information about endpoints and discusses some issues relating to asynchronous processing that specifically affect endpoint providers. Then, it explains some issues relating to data transfer that apply to all types of endpoint providers. Finally, it describes how you can implement each mode of service.

No matter what mode of service you want to implement, you must read the sections "Opening and Binding Endpoints," "Obtaining Information About Endpoints," "Handling Events for Endpoints," and "Sending and Receiving Data." After you have read these sections, you can read the section describing the mode of service you are interested in implementing. Table 3-5 shows how some of the Open Transport protocols fit with an endpoint's mode of service. For example, if you want to use ATP, you would need to read the section "Using Connectionless Transaction-Based Service," beginning on page 3-48. If you want to use ADSP, you would need to read the section "Establishing and Terminating Connections," beginning on page 3-27 and the section "Using Connection-Oriented Transactionless Service," beginning on page 3-44.

Table 3-5 The Open Transport mode-of-service matrix and some Open Transport protocols

	Connectionless	**Connection-oriented**
Transactionless	DDP PPP IP UDP	Serial connection ADSP PAP TCP
Transaction-based	ATP	ASP

Note
The sections that follow present information in such a way as to suggest that communication is always taking place between two Open Transport clients. This does not have to be true. For example, an Open Transport client using a connectionless transactionless DDP endpoint can communicate seamlessly with a client using AppleTalk's DDP protocol and interface. However, because this book is about Open Transport, we always show how communication works between two Open Transport clients. ◆

Opening and Binding Endpoints

Before you can open and bind an endpoint, you must have initialized Open Transport and determined what the endpoint configuration is going to be. Then, you can open and bind the endpoint. You open the endpoint with the `OTOpenEndpoint` or `OTAsyncOpenEndpoint` functions. Opening an endpoint with the `OTOpenEndpoint` function sets the default mode of execution to be synchronous; opening an endpoint with the `OTAsyncOpenEndpoint` function sets the default mode of execution to be asynchronous. You can change an endpoint's mode of execution at any time by calling the `OTSetSynchronous` or `OTSetAsynchronous` function, which are described in the chapter "Providers" in this book.

One of the parameters that you pass to the function used to open the endpoint is a pointer to a configuration structure that Open Transport needs to define the protocol stack providing data transport services. You can use the same configuration for more than one endpoint; however, if you do so, you must use the `OTCloneConfiguration` function to get a valid copy of the configuration structure. The chapter "Configuration Management," in this book, contains information about creating a configuration structure for an endpoint provider.

If you use the `OTAsyncOpenEndpoint` function to open an endpoint, you also specify the entry point to a notifier function that the endpoint provider can use to call your application when an asynchronous or completion event takes place. If you use the `OTOpenEndpoint` function to open an endpoint, and you want to handle asynchronous events using a notifier function, you must use the `OTInstallNotifier` function to install your notifier function. The `OTInstallNotifier` function is described in the chapter "Providers" in this book.

Opening an endpoint also sets up a private data structure used by Open Transport to manage the endpoint provider's operations. This data structure contains information about

- the endpoint's mode of operation, mode of execution, and mode of service
- the size of internal buffers used for sending data and receiving data
- the size of normal transport service data units (TSDUs) and expedited transport service data units (ETSDUs) or, in the case of transactions, the size of replies and requests
- the maximum size of buffers used to hold address and option information for the endpoint
- default option values for the endpoint

Some of this information is private; the rest can be retrieved by calling functions that return information about the endpoint. These functions are described in the next section, "Obtaining Information About Endpoints."

When the function you use to open the endpoint returns, it passes back to you an endpoint reference. You must pass this reference as a parameter to any endpoint provider function or any general provider function. For example, you pass this reference as a parameter to the `OTBind` function, which you must use to bind an endpoint after opening it. Binding an endpoint associates the endpoint with a logical address. Depending on the protocol you use and on your application's needs, you can select a specific address or you can have the protocol choose an address for you. For information about valid address formats, consult the documentation for your protocol. The general rule for binding endpoints is simple: you cannot bind more than one connectionless endpoint to a single address. You can bind more than one connection-oriented endpoint to the same address; for additional information about this possibility, see the section "Processing Multiple Connection Requests" on page 3-33.

No matter what mode of service you need to implement, you must know how to obtain information about the endpoints you have opened and how to handle asynchronous and completion events for these endpoints. These issues are addressed in the next two sections, "Obtaining Information About Endpoints" and "Handling Events for Endpoints." After you read these sections, you can proceed by reading about the mode of service you want to implement.

Obtaining Information About Endpoints

You can use endpoint functions to obtain information about an endpoint's mode of service, state, or address. You can also call general provider functions to determine an endpoint's mode of execution and mode of operation:

- To obtain information about an endpoint's mode of execution, you call the `OTIsSynchronous` function. The function returns a positive integer to indicate that the endpoint is in synchronous mode, or 0 if it is in asynchronous mode.
- To obtain information about an endpoint's mode of operation, you call the `OTIsNonBlocking` or the `OTIsAckingSends` functions.

The `TEndpointInfo` structure contains most of the information you need to determine how you can use an endpoint. This structure specifies the maximum size of the buffers you need to allocate when calling functions that return address and option information or data, and it also contains more specific details about the mode of service the endpoint provides. For example, if you have opened a connection-oriented endpoint, the `servtype` field of the `TEndpointInfo` structure specifies whether the endpoint supports orderly release. You can obtain a pointer to this structure when you open the endpoint, when you bind the endpoint, or when you call the `OTGetEndpointInfo` function.

To obtain information about an endpoint's state, you call the `OTGetEndpointState` function. This function returns a positive integer indicating the endpoint state or a negative integer corresponding to a result code. Table 3-2 on page 3-14 lists and describes endpoint states. If the endpoint is in asynchronous mode and you are not using a notifier function, you might be able to use the `OTGetEndpointState` function to poll the endpoint and determine whether a specific function has finished executing. The completion of some functions result in an endpoint's changing state. For additional information, see Table 3-3 on page 3-18.

To obtain address information about an endpoint or its peer, you can use one of the following two functions:

- `OTGetProtAddress`, which returns the address to which the endpoint is bound. If the endpoint is connection-oriented and currently connected, this function also returns the address to which the endpoint is connected.
- `OTResolveAddress`, which returns the lowest-layer protocol address that corresponds to the name of the endpoint. If you are looking up the address that corresponds to a single name, you can use this function rather than having to open the mapper provider and use the mapper function `OTLookUpName`.

For information about the address formats for the protocol you are using, please consult the documentation supplied for the protocol. For information about obtaining the addresses that correspond to a name pattern, see the chapter "Mappers" in this book.

Handling Events for Endpoints

The section about modes of execution in the chapter "Providers" describes the functions you use to determine what a provider's mode of execution is and to change that mode if needed. It also discusses the special problems that might arise in asynchronous processing and recommends ways of handling these problems.

Like other providers, endpoint providers can operate synchronously or asynchronously. When possible, you should use endpoints in asynchronous mode. If you do, you need to create a notifier function that the provider can call to inform you when an asynchronous function has completed or when an asynchronous event has arrived. Event handling for endpoints is basically the same as that described for providers in the chapter "Providers." One slight difference lies in the way the endpoint provider generates `T_DATA`, `T_EXDATA`, and `T_REQUEST` asynchronous events, which signal the arrival of incoming data or of an incoming transaction request. For the sake of efficiency, the provider notifies you just once that incoming data has arrived. To read all the data, you must call the function that clears the event until the function returns with the `kOTNoDataErr` result. For information about which functions to use to clear these events, see Table 3-8 on page 3-27.

You do not have to issue these calls in the notification routine itself, but until you make the consuming calls and receive a `kOTNoDataErr` error, another `T_DATA`, `T_EXDATA`, or `T_REQUEST` event will not be issued. You 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 or commands to deliver to the client. If the queue is empty when the data or 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 or command is queued. Instead, whenever the client reads the data or 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.

Not all endpoint functions are affected by an endpoint's mode of execution. Those functions that do behave differently when they are executed asynchronously are listed in Table 3-6. For each function, the table lists the corresponding completion event.

Table 3-6 Endpoint functions that behave differently in synchronous and asynchronous mode

Function	Completion event
`OTOptionManagement`	`T_OPTIONMANGEMENTCOMPLETE`
`OTBind`	`T_BINDCOMPLETE`
`OTUnbind`	`T_UNBINDCOMPLETE`
`OTAccept`	`T_ACCEPTCOMPLETE`
`OTSndRequest`	`T_REQUESTCOMPLETE`
`OTSndReply`	`T_REPLYCOMPLETE`
`OTSndURequest`	`T_REQUESTCOMPLETE`
`OTSndUReply`	`T_REPLYCOMPLETE`
`OTDisconnect`	`T_DISCONNECTCOMPLETE`
`OTGetProtAddress`	`T_GETPROTADDRCOMPLETE`
`OTResolveAddress`	`T_RESOLVEADDRCOMPLETE`

For compatibility with the XTI standard, Open Transport also includes the endpoint provider function `OTLook`. You can use the `OTLook` function

- to poll for asynchronous events, like incoming data or connection requests
- to determine the cause of a `kOTLookErr` result

  Asynchronous functions can return this result. In addition, asynchronous events that require immediate attention can cause some synchronous functions to fail with the `kOTLookErr` result. In this case, you can call the `OTLook` function to determine the event that caused the function to fail. Table 3-7 lists the functions that can return the result `kOTLookErr` when the corresponding event is pending.

Table 3-7 Pending asynchronous events and the synchronous functions they can affect

Function that fails	Pending events
`OTAccept, OTConnect`	`T_DISCONNECT, T_LISTEN`
`OTListen, OTRcvConnect, OTRcvOrderlyDisconnect, OTSndOrderlyDisconnect,OTSndDisconnect`	`T_DISCONNECT`
`OTRcv, OTRcvRequest, OTRcvReply, OTSnd, OTSndRequest, OTSndReply`	`T_DISCONNECT, T_ORDREL`
`OTRcvUData, OTSndUData`	`T_UDERR`
`OTUnbind`	`T_LISTEN, T_DATA`

Having used the `OTLook` function to determine what asynchronous event caused your function to fail, you must call one of the functions listed in Table 3-8 to clear the event, and then you can retry the function that failed.

Table 3-8 Pending asynchronous events and the functions that clear them

Pending event	Open Transport function that clears the event
`T_LISTEN`	`OTListen`
`T_CONNECT`	`OTRcvConnect`
`T_DATA`	`OTRcv, OTRcvUData`
`T_EXDATA`	`OTRcv`
`T_DISCONNECT`	`OTRcvDisconnect`
`T_UDERR`	`OTRcvUDErr`
`T_ORDREL`	`OTRcvOrderlyDisconnect`
`T_GODATA`	`OTSnd, OTSndUData`
`T_GOEXDATA`	`OTSnd`

Establishing and Terminating Connections

To implement a connection-oriented service, you must complete the following steps:

- establish a connection
- process any data associated with establishing the connection if this is permitted for the endpoint
- transfer data
- terminate the connection when you are finished transferring data

The following sections explain how you establish and terminate a connection. The functions you use to establish and terminate a connection are the same for transactionless as for transaction-based service. The calls you use to transfer data differ depending on which mode of service you choose—transactionless or transaction-based. The section "Using Connection-Oriented Transactionless

Service" on page 3-44 explains how you transfer data once you have established a connection. In the text that follows, *active peer* refers to the endpoint initiating a connection; *passive peer* refers to the endpoint accepting a connection request.

Before you can use a connection-oriented endpoint to initiate or accept a connection, you must open and bind the endpoint. For example, if you are using AppleTalk, you might open an ADSP endpoint, which offers connection-oriented transactionless service. You don't have to do anything special to bind an endpoint that is intended to be the active peer of a connection. However, when you bind an endpoint intended to be the passive peer of a connection, you must specify a value for the `qlen` field of the `reqAddr` parameter for the `OTBind` function. The `qlen` field indicates the number of outstanding connection requests that can be queued for that endpoint. Note that the value you specify indicates the desired value. Open Transport might negotiate a lower value, depending upon the number of internal buffers available. The negotiated value of outstanding connection indications is returned to you in the `qlen` field of the `retAddr` parameter for the `OTBind` function.

You are allowed to bind multiple connection-oriented endpoints to a single address. However, only one of these endpoints can accept incoming connection requests. That is, only one endpoint can specify a value for `qlen` that is greater than 0. For more information, see the section "Processing Multiple Connection Requests" on page 3-33.

Establishing a Connection

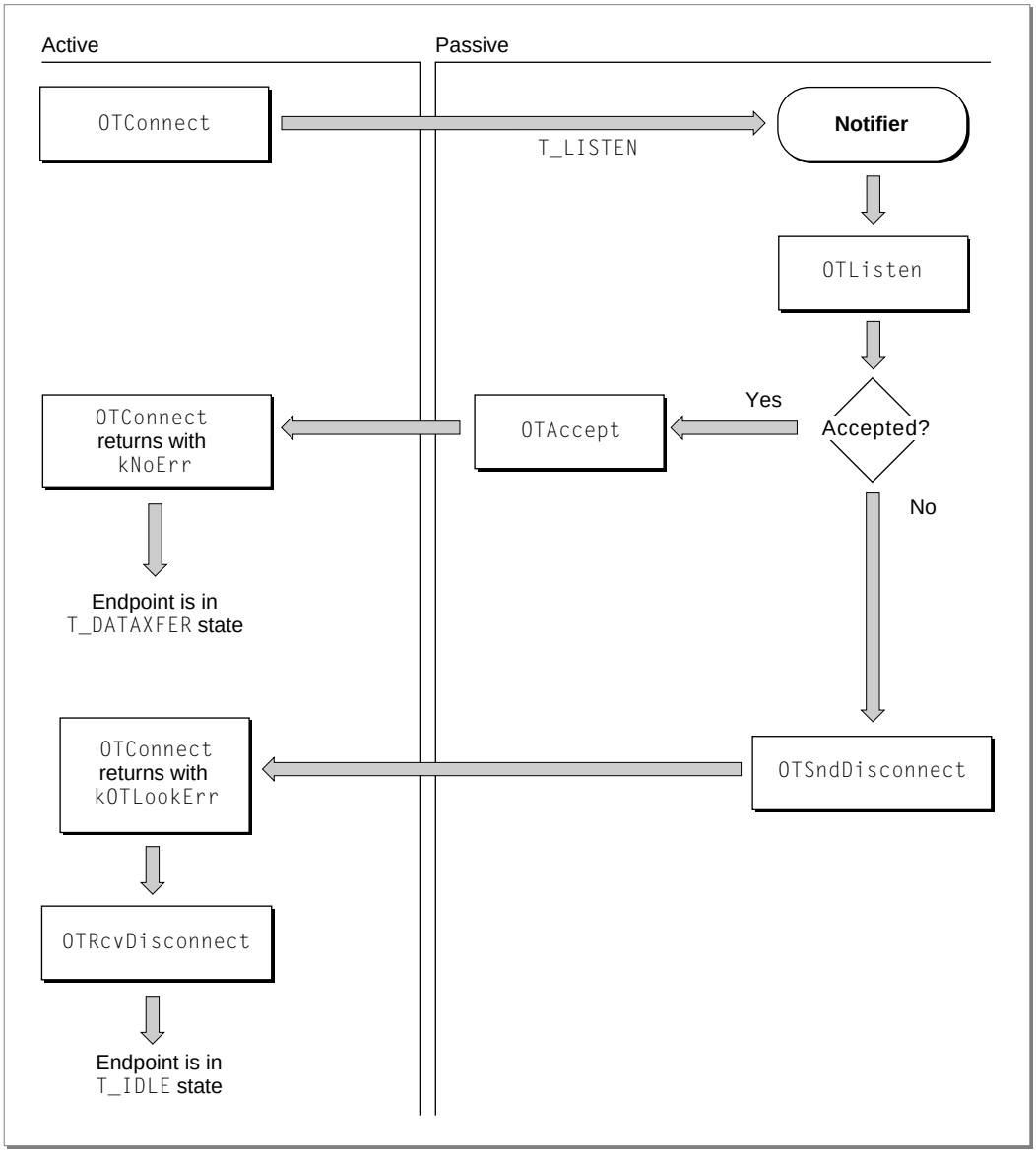
You use the following functions to establish a connection:

Active peer calls	Passive peer calls	Meaning
`OTConnect`		Requests a connection to the passive peer.
	`OTListen`	Listens for an incoming connection request.
	`OTAccept`	Accepts the connection request identified by the `OTListen` function. The connection can be accepted by a different endpoint than the one listening for incoming connection requests.

Active peer calls	Passive peer calls	Meaning
`OTRcvConnect`		Reads the status of a pending or completed asynchronous call to the `OTConnect` function.
	`OTSndDisconnect`	Rejects an incoming connection request.
`OTRcvDisconnect`		Identifies the cause of a rejected connection and acknowledges the corresponding disconnection event.

Figure 3-3 illustrates the process of establishing a connection in synchronous mode.

Figure 3-3 Establishing a connection in synchronous mode

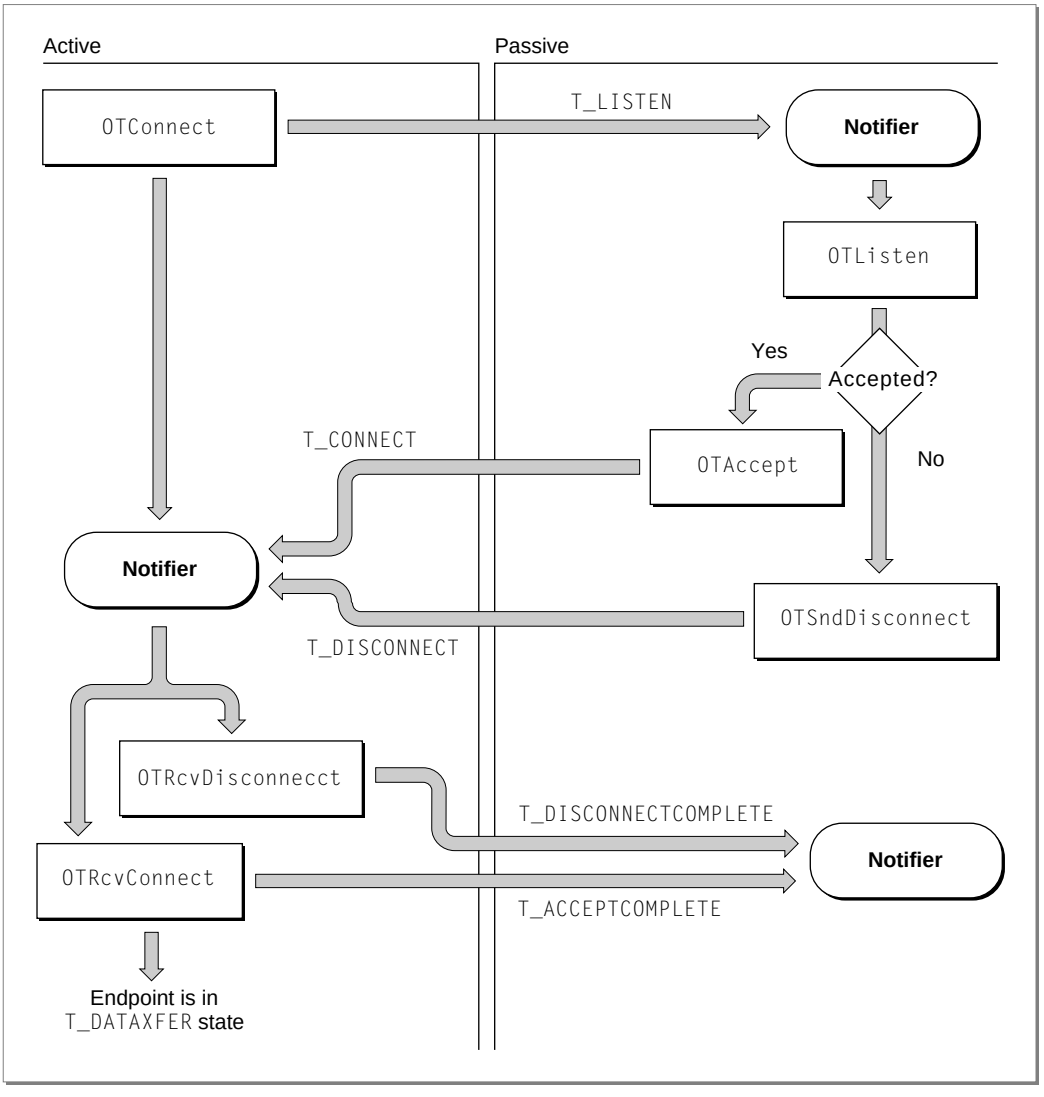


As Figure 3-3 shows, if the active peer is in synchronous mode, the `OTConnect` function does not return until the connection has been established or the connection attempt has been rejected. If the passive peer has a notifier function installed, the endpoint provider calls it, passing `T_LISTEN` for the `code` parameter. The notifier calls the `OTListen` function, which reads the connection request. The passive peer can now either accept the connection request using the `OTAccept` function or reject the request by calling the `OTSndDisconnect` function. The connection attempt might also fail if the request is never received and the endpoint provider times out the call to the `OTConnect` function.

If the passive peer calls the `OTAccept` function to accept the connection, the `OTConnect` function returns with `kNoErr`. If the passive peer rejects the connection by executing the `OTSndDisconnect` function or the request is timed out, the `OTConnect` function returns with `kOTLookErr`. When the `OTConnect` function returns, the active peer must examine the result and, depending on the outcome, either begin to transfer data if the function succeeds or call the `OTRcvDisconnect` function if the function fails. The active peer must call the `OTRcvDisconnect` function to restore the endpoint to a valid state for subsequent operations. Note that even though the passive peer is in a synchronous state, you can use a notifier function to be called in case of a `T_LISTEN` event. Alternately, you could also use the `OTLook` function to poll the passive endpoint for a `T_LISTEN` event.

If the active peer is in asynchronous mode, the `OTConnect` function returns right away, and the active peer must rely on its notifier function to determine whether the call succeeded. Figure 3-4 illustrates the process of establishing a connection when the active peer is in asynchronous mode.

Figure 3-4 Establishing a connection in asynchronous mode



The active peer calls the OTConnect function, which returns right away with a code of kOTNoError. The endpoint provider calls the passive peer's notifier, passing T_LISTEN for the code parameter. If the passive peer accepts the connection, the endpoint provider calls the active peer's notifier, passing T_CONNECT for the code parameter.

If the passive peer rejects the connection or if the connection times out, the endpoint provider calls the active peer's notifier, passing T_DISCONNECT for the code parameter. The active peer must then call either the OTRcvConnect function in response to a T_CONNECT event or the OTRcvDisconnect function in response to a T_DISCONNECT event. The endpoint provider, in turn, passes the T_ACCEPTCOMPLETE event back to the passive peer (for a successful connection) or the T_DISCONNECTCOMPLETE event (for a failed connection). The passive peer requires the information provided by these two events to determine whether the connection succeeded.

Sending User Data With Connection or Disconnection Requests

The OTConnect function and the OTSndDisconnect function both pass data structures that include fields for data that you might want to send at the time that you are setting up or tearing down a connection. However, you can only send data when calling these two functions if the connect and discon fields of the TEndpointInfo structure specify that the endpoint can send data with connection or disconnection requests. The amount of data sent must not exceed the limits specified by these two fields. To determine whether the endpoint provider for your endpoint supports data transfer during the establishment of a connection, you must examine the connect and discon fields of the TEndpointInfo structure for the endpoint.

Processing Multiple Connection Requests

If you process multiple connection requests for a single endpoint, you must make sure that the number of outstanding connection requests does not exceed the limit defined for the listening endpoint when you bound that endpoint. An outstanding connection request is a request that you have read using the OTListen function but that you have neither accepted nor rejected. You must also decide whether to accept connections on the same endpoint that is listening for the connections or on a different endpoint.

When you bind the passive endpoint, you must specify a value greater than 0 for the qlen field of the reqAddr parameter to the OTBind function. This value indicates the number of outstanding connections that the provider can queue

for this endpoint. Note that Open Transport might negotiate this number to a lower value. If it does, the negotiated value is returned in the `qlen` field of the `retAddr` parameter when the `OTBind` function returns. As you process incoming connection requests, you must check that the number of connections still waiting to be processed does not exceed this negotiated value for the listening endpoint. How you do this depends on the number of outstanding requests and on whether you are accepting connection requests on the same endpoint as the endpoint listening for requests or accepting them on a different endpoint. Connection acceptance is governed by the following rules:

- You can bind more than one connection-oriented endpoint to the same address, but you can use only one of these endpoints to listen for connection requests.
- If you accept a connection on the same endpoint that is listening for connection requests, you must have responded to all previous connection requests received on the endpoint using `OTAccept` or `OTSndDisconnect` functions. Otherwise, the `OTAccept` function fails. If you have not responded to all previous connection requests, you should accept the connection on a different endpoint.

  If you accept a connection on the same endpoint that received the connection request and there are outstanding connection or disconnection indications for that endpoint, the `OTAccept` function fails.
- If you accept a connection on an endpoint that is different from the endpoint that received the connection request, you do not have to bind the endpoint to which you are passing off the connection. If the endpoint is not bound, the endpoint provider automatically binds it to the address of the endpoint that listened for the connection request.

  If you choose to explicitly bind the endpoint accepting the connection to the address of the endpoint listening for the connection, you must set the `qlen` field of the `reqAddr` parameter to the `OTBind` function to 0.

What these rules add up to in practical terms is that if you anticipate managing more than one connection at a time, you should open an endpoint to listen for connections and then open additional endpoints as needed to accept incoming connections. The decision of whether to bind the additional endpoints to the same address or to a different address is affected only by the availability of endpoints to your application.

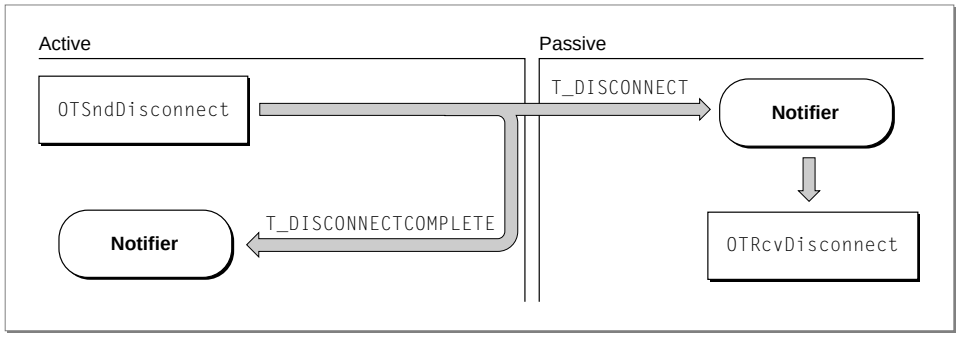
Terminating a Connection

You can terminate a connection using either an abortive or orderly disconnect. During an abortive disconnect, the connection is torn down without the underlying protocol taking any steps to make sure that data being transferred has been sent and received. When the client calls the `OTSndDisconnect` function, the connection is immediately torn down, and the client cannot be sure that the provider actually sent any locally buffered data. During an orderly disconnect, the underlying protocol ensures at least that all outgoing data is actually sent. Some protocols go further than this, using an over-the-wire handshake that allows both peers to finish transferring data and agree to disconnect. The following sections describe the steps required for abortive and orderly disconnects.

Using an Abortive Disconnect

You use the `OTSndDisconnect` and `OTRcvDisconnect` functions to perform an abortive disconnect. Figure 3-5 illustrates the process for two asynchronous endpoints. The figure shows the active peer initiating the disconnection; in fact, either endpoint can initiate the disconnection.

Figure 3-5 An abortive disconnect



In asynchronous mode, the endpoint initiating the disconnection calls the `OTSndDisconnect` function. Parameters to the function identify the endpoint and point to a `TCall` structure that is only of interest if the endpoint provider supports sending data with disconnection requests. To determine whether your protocol does, you must examine the value of the `discon` field of the `TEndpointInfo` structure for your endpoint. If you do not want to send data or if you cannot send data to the passive peer, you can set `TCall` to a `NULL` pointer.

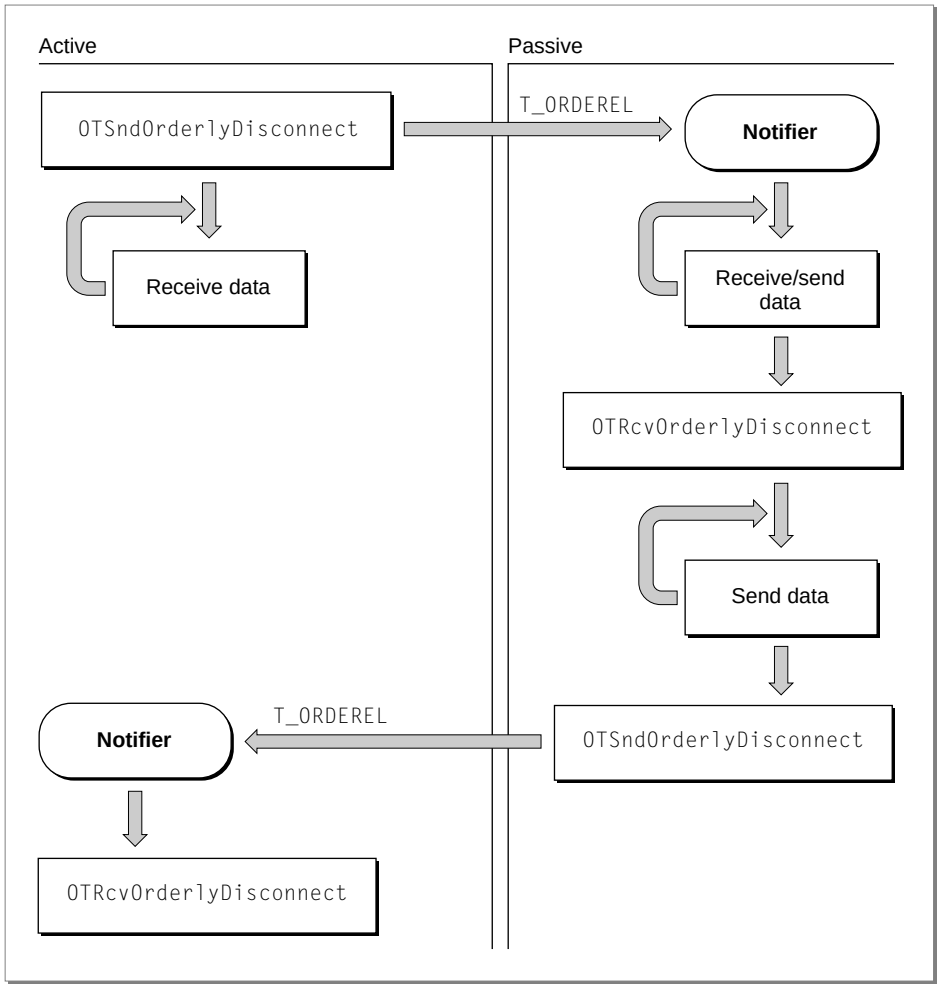
The endpoint provider receiving the disconnect request calls the passive peer's notifier function, passing `T_DISCONNECT` for the `code` parameter. The client must acknowledge the disconnection event by calling the function `OTRcvDisconnect`. This function clears the event and retrieves any data sent with the event. Parameters to the `OTRcvDisconnect` function identify the endpoint sending the disconnection and point to a `TDiscon` structure that is only of interest if the endpoint provider supports sending data with disconnection requests or if the passive peer is managing multiple connections and needs to inform the active peer which of the connections has been closed by using the `sequence` field of the `TDiscon` structure. Otherwise, you can set `TDiscon` to a `NULL` pointer. When the connection has been closed, the endpoint provider calls the active peer's notifier, passing `T_DISCONNECTCOMPLETE` for the `event` parameter. At this time the endpoint is once more in the `T_IDLE` state.

Using Orderly Disconnects

There are two kinds of orderly disconnects: remote orderly disconnects and local orderly disconnects. The first kind, supported by TCP, provides an over-the-wire (three-way) handshake that guarantees that all data has been sent and that both peers have agreed to disconnect. The second kind, supported by ADSP and most other connection-oriented transactionless protocols, is a locally implemented orderly release mechanism ensuring that data currently being transferred has been received by both peers before the connection is torn down. To determine whether your protocol supports orderly disconnects, you must examine the `servtype` field of the `TEndpointInfo` structure for the endpoint. A value of `T_COTS_ORD` or `T_TRANS_ORD` indicates that the endpoint supports orderly release. It is safest to assume, unless you know for certain it to be otherwise, that the endpoint supports only local orderly disconnects.

Figure 3-6 shows the steps required to complete a remote orderly disconnect. The figure shows the active peer initiating the disconnection; in fact, either peer can initiate the disconnection.

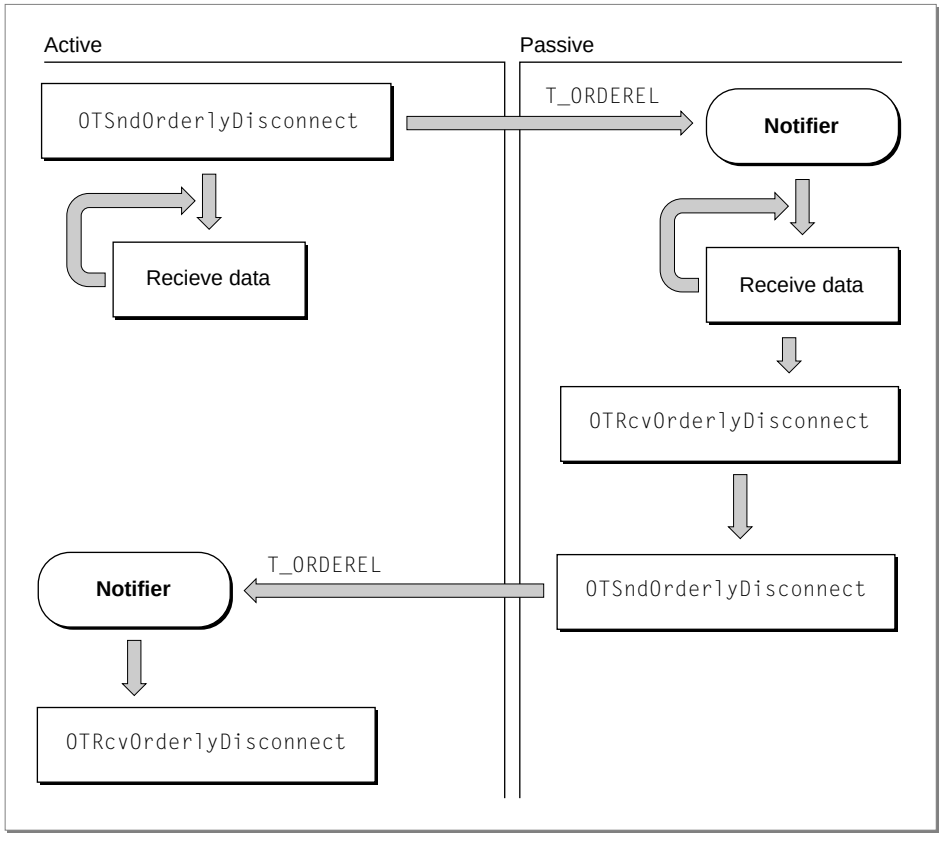
Figure 3-6 Remote orderly disconnect



The active peer initiates the disconnection by calling the `OTSndOrderlyDisconnect` function to begin the process and to let the remote endpoint know that the active peer will not send any more data. (Following the call to this function, the active peer can receive data but it cannot send any more data.) The provider calls the passive peer's notifier function, passing `T_ORDREL` for the `code` parameter. In response, the passive peer must read any unread data and can send additional data. After it has finished reading the data, it must call the `OTRcvOrderlyDisconnect` function to acknowledge receipt of the orderly release indication. After calling this function, the passive peer must not attempt to read any more data; however, it can continue to send data. When the passive peer is finished sending any additional data, it must call the `OTSndOrderlyDisconnect` function to complete its part of the disconnection. Following this call, it cannot send any data. The endpoint provider calls the active peer's notifier, passing `T_ORDREL` for the `code` parameter, and the active peer calls the `OTRcvOrderlyDisconnect` function to acknowledge receipt of the disconnection event and to place the endpoint in the `T_IDLE` state if this was the only outstanding connection.

Figure 3-7 shows the steps required to complete a local orderly disconnect.

Figure 3-7 A local orderly disconnect



As you can see, the sequence of steps is very similar to that shown in Figure 3-6. The main difference is that the connection is broken as soon as the active peer calls the `OTSndOrderlyDisconnect` function. As a result, either peer can continue to read any unread data, but neither peer can send data after the initial call to the `OTSndOrderlyDisconnect` function.

Sending and Receiving Data

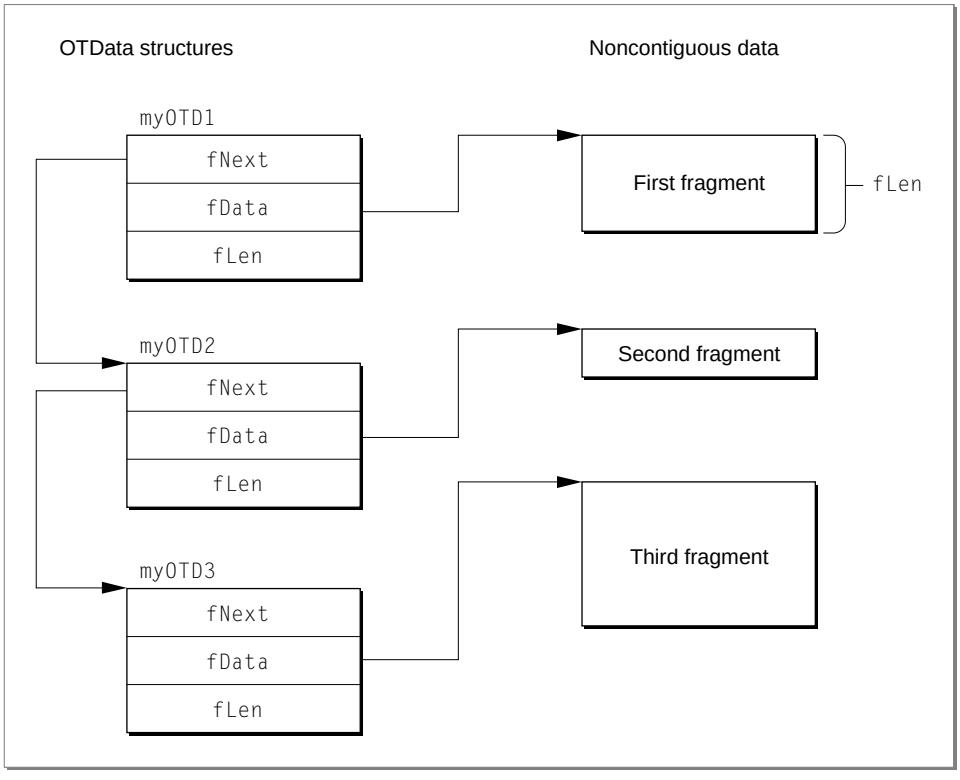
This section describes some of the issues that affect send and receive operations inasmuch as these issues affect every type of endpoint. After you read this section, you can read whichever section describes the type of data transfer you are interested in.

Sending Noncontiguous Data

When sending data, you normally use a `TNetbuf` structure to specify the location and size of the buffer containing the data to be sent. Open Transport also allows you to send data that is not contiguous; however, you need to use a different structure to specify the location of the data fragments in memory. This structure is called the `OTData` structure.

Figure 3-8 shows how you use `OTData` structures to describe noncontiguous data. The first structure, `myOTD1`, contains information about the first data fragment: the `fData` field contains the starting address of the fragment, and the `fLen` field contains the length of the fragment. The field `fNext` contains the address of the `OTData` structure, `myOTD2`, which specifies the size and location of the second fragment. In turn, the structure `myOTD2`, contains the address of the `OTData` structure that specifies the location and size of the third fragment. You must set the `fNext` field of the `OTData` structure used to describe the last data fragment to `NULL`.

Figure 3-8 Describing noncontiguous data



Sending Data Using Multiple Sends

If you are sending a data unit using multiple sends, you must do the following:

1. Set the `T_MORE` bit in the flags field each time you call the function. This lets the provider know that it has not yet read the entire data unit.
2. Clear the `T_MORE` bit the last time you call the function. This lets the provider know that the data unit is complete.

Even though you are using multiple sends to send the data, the total size of the data sent cannot exceed the value specified for the `tsdu` field (for normal data or replies) or `etsdu` field (for expedited data or requests) of the `TEndpointInfo` structure for the endpoint.

Sending data using multiple sends does not necessarily affect the way in which the remote client receives the data. That is, just because you have used several calls to a send function to send data does not mean that the remote client must call a receiving function several times to read the data.

Receiving Data

If you are reading data and if the `T_MORE` bit in the flags field is set, this means that the buffer you have allocated to hold the data is not big enough. You must copy the data you have already received to a different buffer and then call the receive function again to read more data until the `T_MORE` bit is cleared, which indicates that you have read the entire data unit.

No-Copy Receiving

Open Transport allows you to receive data without doing the extra copying that is normally involved in receiving data, which can save time and resources. For example, you might have received some data that needs to be written to disk and you have four files, each with a different buffer, that are expecting data. Normally what you would do is store the data into a temporary buffer while you determined which of the four files was the right destination. When you identified the target, you'd then copy the data from the temporary buffer into that file's buffer.

A no-copy receive allows you to peek at the data when you receive it and write it out immediately. Open Transport does this by giving you access to a special no-copy receive buffer, `OTBuffer`. To take advantage of this buffer, it is absolutely crucial that you

- don't touch it
- release it quickly
- only release it once; don't release it multiple times

▲ **WARNING**
The no-copy receive buffer is read-only and you must **never** under any circumstances attempt to write to it. if you write to it, you can crash the system. ▲

You need to release the no-copy receive buffer (with the `OTReleaseBuffer` function) as soon as you are finished using it so that are not tying up system resources required elsewhere. If you hold onto the buffer, one consequence is

that your Ethernet driver starts making its own copies as it receives more data, and if it isn't well designed, it may run out of space and lose packets.

The no-copy receive buffer is actually a linked chain of buffers, with the next buffer pointed to by the `fNext` field in each buffer. You can access all of the received data by tracing the chain of `fNext` pointers. For your convenience, Open Transport provides the `OTBufferInfo` structure and the utility functions, `OTReadBuffer` and `OTBufferDataSize`, to read through the `OTBuffer` structure.

Transferring Data Efficiently

Some protocols support XTI-level options that you can use to change the size of Open Transport's internal send and receive buffers and to change the size of the "low-water mark" that Open Transport uses to determine how much data should accumulate in these buffers before it sends the data or lets the client know that data has arrived. If your protocol supports these options, you can reset these values to fit your application's needs. For more information, see the section describing XTI-level options in the chapter "Option Management" in this book.

Transferring Data Between Transactionless Endpoints

Open Transport defines two sets of functions that you can use to send and receive data. You use one set with connectionless service and the other with connection-oriented service.

Using Connectionless Transactionless Service

You use connectionless transactionless service, as provided by AppleTalk's DDP for example, to send and receive discrete data packets. Most often applications use higher-level protocols that depend, in turn, upon more basic protocols that use connectionless transactionless service. For example, all of AppleTalk's higher-level protocols make use of DDP to send and receive data.

After opening and binding a connectionless transactionless endpoint, you can use three functions to send and receive data:

- the `OTSndUData` function to send data
- the `OTRcvUData` function to receive data
- the `OTRcvUDErr` function to determine why a send operation did not succeed

Either endpoint can send or receive data. However, the endpoint sending data cannot determine whether the other endpoint has actually received the data.

Some endpoints are not able to determine that the specified address or options are invalid until after the data is sent. In this case, the sender's endpoint provider issues the `T_UDERR` event. You should include code in your notifier function that calls the `OTRcvUDErr` function in response to this event to determine what caused the send function to fail and to place the sending endpoint in the correct state for further processing.

If the endpoint receiving data has allocated a buffer that is too small to hold the data, the `OTRcvUData` function returns with the `T_MORE` bit set in the `flags` parameter. In this case, you should call the `OTRcvUData` function repeatedly until the `T_MORE` bit is cleared.

Using Connection-Oriented Transactionless Service

You use connection-oriented transactionless service, such as provided by ADSP, to exchange full-duplex streams of data across a network. Connection-oriented transactionless endpoints use the `OTSnd` function to send data and the `OTRcv` function to receive data. Either endpoint can call either of these functions. Parameters to the `OTSnd` function identify the endpoint sending the data, the buffer that holds the data, the size of the data, and a `flags` value that specifies whether the data sent is normal or expedited and whether multiple sends are being used to send the data. Parameters to the `OTRcv` function identify the receiving endpoint, the area in memory where the data should be copied, the size of the data, and a `flags` value that specifies whether the client needs to call `OTRcv` more than once to retrieve the data being sent.

Some endpoints support the use of expedited data, and some support the use of separators to break the data stream into logical units. You need to examine the endpoint's `TEndpointInfo` structure to determine if the endpoint supports either of these features:

- The `etsdu` field of the `TEndpointInfo` structure specifies whether the endpoint supports the use of expedited data and, if so, specifies its size. For example, ADSP supports the use of expedited data to send attention messages between peer endpoints. In general, it is recommended that you do not use expedited data because doing so results in code that is not transport independent.
- The `tsdu` field of the `TEndpointInfo` structure specifies the maximum size of normal data that the endpoint can send or receive. In those cases where the

endpoint supports the breaking up of the data stream into logical units, the TSDU size specifies what the maximum size of any such unit may be.

IMPORTANT

Values for the `tsdu` and `etsdu` fields of the `TEndpointInfo` structure that are returned when you open an endpoint might change after the endpoint is connected because the endpoint providers can negotiate different values when establishing a connection. If the endpoint supports variable maximum limits for TSDU and ETSDU size, you should call the `OTGetEndpointInfo` function after the connection has been established to determine what the current limits are. ▲

To send expedited data, you must set the `T_EXPEDITED` bit in the `flags` parameter. If the receiving client is in the middle of reading normal data and the `OTRcv` function returns expedited data, the next `OTRcv` that returns without `T_EXPEDITED` set in the flags field resumes the sending of normal data at the point where it was interrupted. It is the responsibility of the client to remember where that was.

There are several ways of breaking up a data stream into logical size units.

- If the endpoint supports it, enable the use of the `T_MORE` flag to the `OTSnd` function. For example, using ADSP, you can do this by setting the `EOM` option when you connect the endpoints. Sending data with the `T_MORE` bit set informs the receiving endpoint that the TSDU is being sent using multiple `OTSnd` calls. When sending the last packet, do not set the `T_MORE` bit. Because these packets are guaranteed to be delivered in the order sent, the receiving endpoint can determine when the last packet has arrived by examining the flags.
- If the endpoint supports it, send a zero-length TSDU to indicate the end of a TSDU. The receiving endpoint needs to test the `nbytes` field of the `OTSnd` function to determine if this is the last transmission. To determine whether the endpoint supports this feature, you need to examine the `flags` field of the `TEndpointInfo` structure; zero-length TSDUs are supported if the `T_SENDZERO` bit is set.
- Use the data transferred with your first send to specify the name and size of the file that you want to send. The receiving endpoint can save the size value and decrement it by the value specified by the `nbytes` parameter of each

subsequent send until the number equals 0. This last method is the only one that is transport-independent.

Transferring Data Between Transaction-Based Endpoints

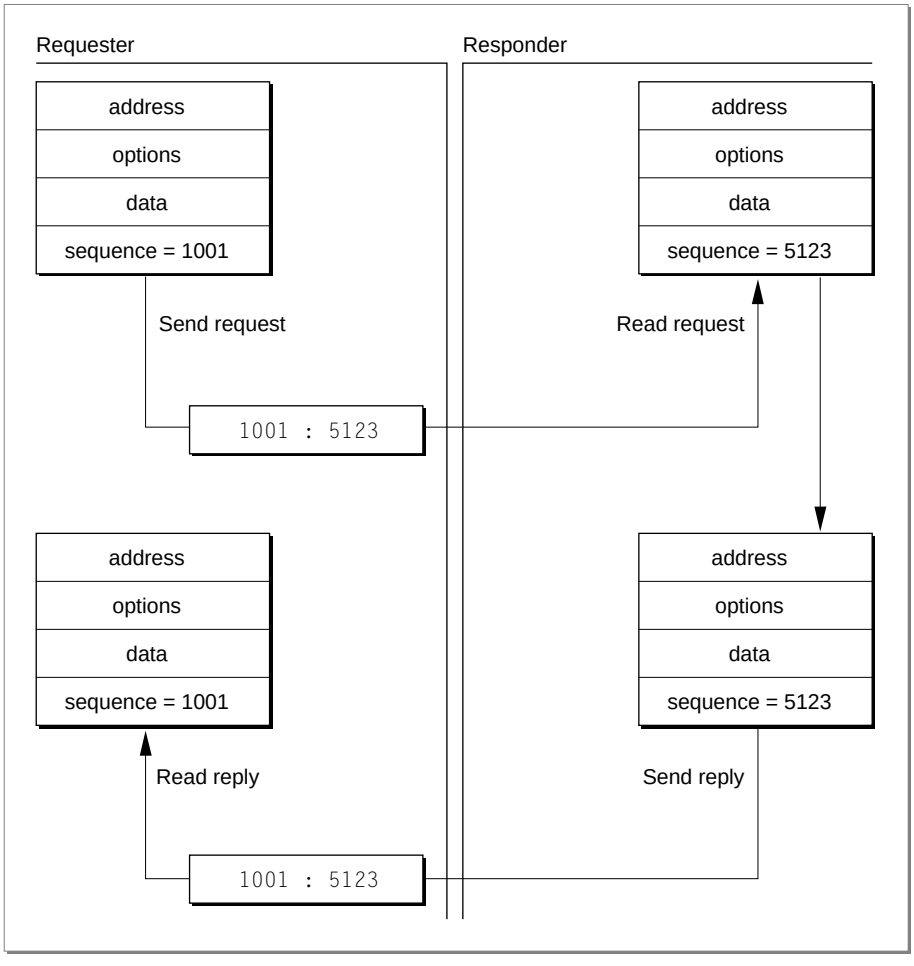
Open Transport defines two sets of functions that you can use to conclude a transaction. One set is defined for connectionless transactions; the other set is defined for connection-oriented transactions. A **transaction** is a process during which one endpoint, the *requester*, sends a request for a service. The remote endpoint, called the *responder*, reads the request, performs the service, and sends a reply. When the requester receives the reply, the transaction is complete.

You can implement applications that use transactions in the following two ways:

- You can write a single application that handles both the requester and responder actions of a transaction and run that application on two networked nodes. This method allows each application to act as either the requester or the responder. Either side can initiate a transaction, but only one side can control the communication during a single transaction.
- You can write two applications, one implementing the requester part of a transaction and the other implementing the responder side. This model lends itself well to a client-server relationship, in which many nodes on a network run the requester application (client), while one or more nodes run the responder application (server); one server can respond to transaction requests from several clients.

Because one endpoint can conduct multiple transactions at any one time, it is crucial that requesters and responders be able to distinguish one transaction from another. This is done by means of a **transaction ID,** a number that uniquely identifies a transaction. Because this is not the same number for the requester as it is for the responder, some explanation is required. Figure 3-9 shows how the transaction ID is generated by the requesting application and the provider during the course of a transaction.

Figure 3-9 How a transaction ID is generated



The requester initiates a transaction by sending a request. The requester passes information about the request in a data structure that includes a `seq` field, which specifies the transaction ID of the request. The requester initializes this field to some arbitrary, unique number. Before sending the request, the endpoint provider saves this number in an internal table and assigns another number to the `seq` field, which it guarantees to be unique for the requester's machine. The endpoint provider also saves the new number along with the

requester-generated sequence number. For example, in Figure 3-9, the requester assigns the number 1001; the endpoint provider assigns the number 5123.

When the responder receives the request, it reads the request information, including the provider-generated sequence number, into buffers it has reserved for the request data. When the responder sends a reply, it specifies the sequence number it read when it received the request.

Before the requester's endpoint provider advises the requester that the reply has arrived, it examines the sequence number of the reply and looks in its internal table to determine which requester-generated sequence number it matches. It then substitutes that number for the sequence number it received from the responder. By using this method Open Transport guarantees that transactions are uniquely identified, and the requester is able to match incoming replies with outgoing requests.

Using Connectionless Transaction-Based Service

You use connectionless transaction-based service, such as provided by ATP, to enable two connectionless endpoints to complete a transaction.

The requester initiates the transaction by calling the `OTSndURequest` function. Parameters to the `OTSndURequest` function specify the destination address, the request data, any options, and a sequence number to identify this transaction. The requester must supply a sequence number if it is sending multiple requests, so that later on it can match replies to requests. The requester can cancel an outgoing request by calling the `OTCancelURequest` function. A requester can implement its own timeout mechanism by installing a Time Manager task and calling the `OTCancelURequest` function after a specific amount of time has elapsed without a response to the request.

If the responder is synchronous and blocking, the `OTRcvURequest` function returns after it has read the request. If the responder is asynchronous or not blocking and has a notifier installed, the endpoint provider calls the notifier, passing `T_REQUEST` for the `code` parameter. When the responder receives this event, it must call the `OTRcvURequest` function to read the request. On return, parameters to the `OTRcvURequest` function specify the address of the requester, option values, the request data, flags information, and a sequence number to

identify the transaction. When the responder sends a reply to the request, it must use the same sequence number for the reply. If the responder's buffer is too small to contain the request, the endpoint provider sets the `T_MORE` bit in the `flags` parameter. The responder must call the `OTRcvURequest` function until the `T_MORE` bit is clear. This indicates that the entire request has been read.

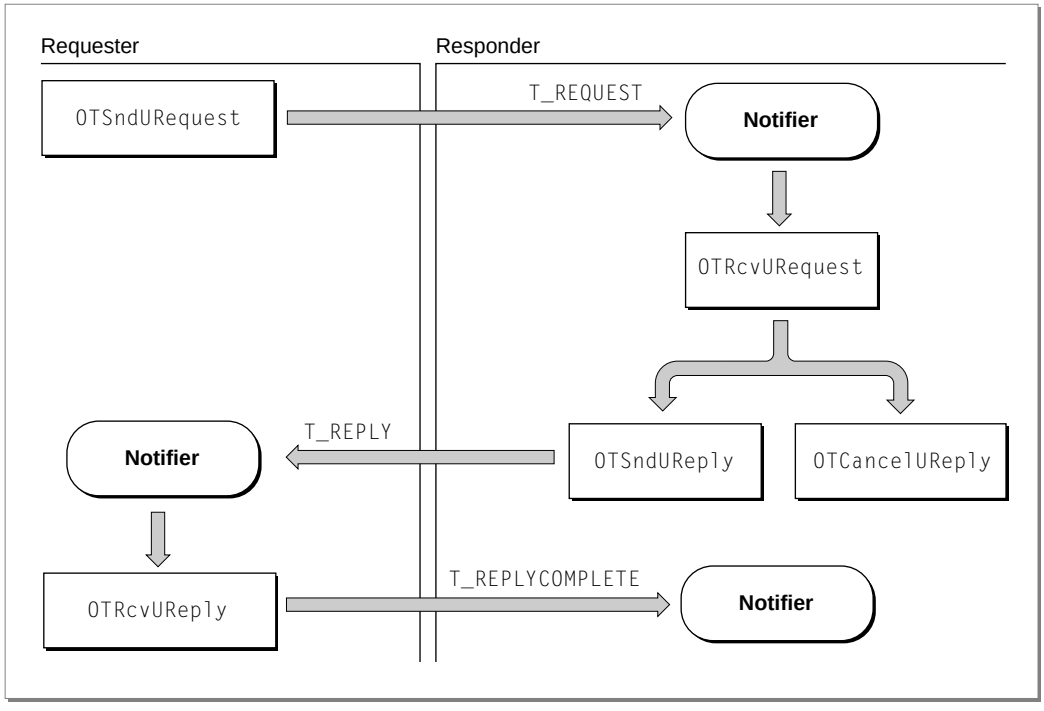
Having read the request, the responder can reply to the request using the `OTSndUReply` function or reject the request using the `OTCancelUReply` function. Although the requester is not advised that the responder has rejected a request, it's important that the responder explicitly cancel an incoming request in order to free memory reserved by the `OTRcvURequest` function.

If the requester is in synchronous blocking mode, the `OTRcvUReply` function waits until a reply comes in. Otherwise, if a notifier is installed, the endpoint provider calls the notifier, passing `T_REPLY` for the `code` parameter. The notifier must call the `OTRcvUReply` function. On return, parameters to the function specify the address of the endpoint sending the reply, specify option values, flag values, reply data, and a sequence number that identifies the request matching this reply. If the `T_MORE` bit is set in the `flags` parameter, the requester has allocated a buffer that is too small to contain the reply data. The requester must call the `OTRcvUReply` function until the `T_MORE` bit is clear; this indicates that the complete reply has been read.

If the request is rejected or fails in some other way, the requester receives the `T_REPLY` event. However, the `OTRcvUReply` function returns with the result `kETIMEDOUTErr`. Otherwise, the only useful information returned by the function is the sequence number of the request that has failed.

Figure 3-10 illustrates how connectionless transaction-based endpoints in asynchronous mode exchange data.

Figure 3-10 Data transfer using connectionless transaction-based endpoints in asynchronous mode



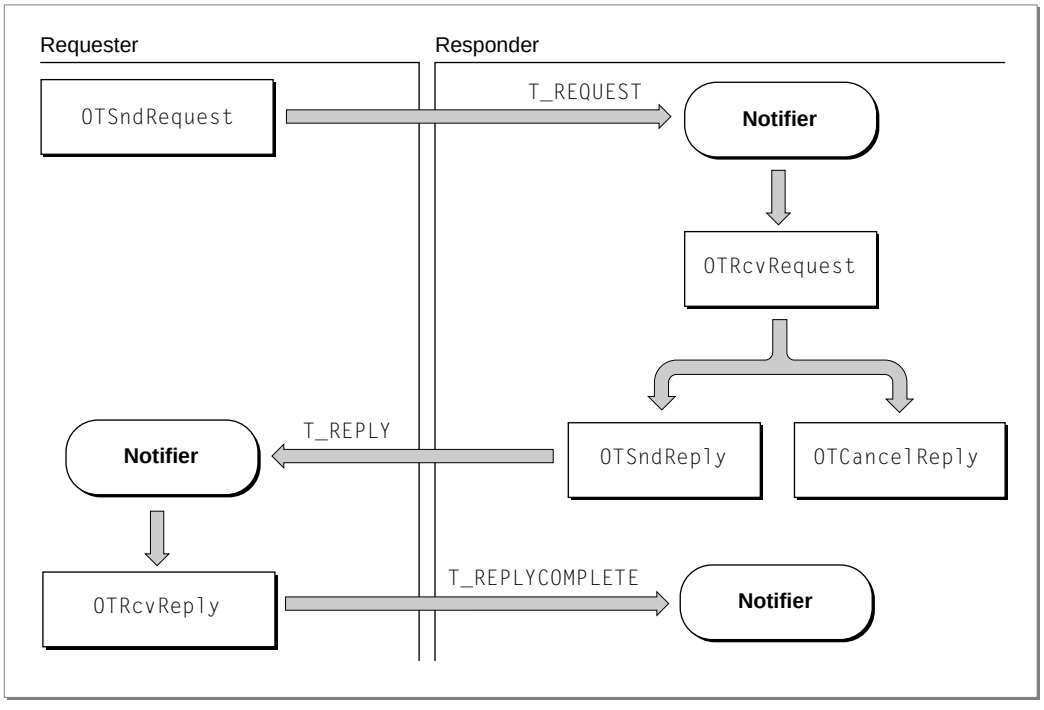
Using Connection-Oriented Transaction-Based Service

Connection-oriented transaction-based endpoints allow you to transfer data in exactly the same way as connectionless transaction-based endpoints except that, because the endpoints are connected, it is not necessary to specify an address when using the functions to send and receive requests and replies. The only other difference is that a connection-oriented transaction may be interrupted by a connection or disconnection request.

The section "Using Connectionless Transaction-Based Service," beginning on page 3-48 describes the sequence of functions used to transfer data using a transaction. Figure 3-11 shows the sequence of functions called during a connection-oriented transaction; both requester and responder are in asynchronous mode. This sequence is the same as for connectionless transaction-based service, as shown in Figure 3-10 on page 3-50. Of course, you

use different functions to complete these two types of transactions: the names of the functions shown in Figure 3-11 do not include a "U" in the function name.

Figure 3-11 Data transfer using connection-oriented transaction-based endpoints in asynchronous model



For information about how to handle disconnection requests that might occur during a transaction, see "Using Orderly Disconnects," beginning on page 3-36.

Endpoints Reference

This section describes the data types and functions that you use with endpoints. You can also use general provider data types and functions with endpoints. General structures and functions are described in the reference section of the chapter "Providers" in this book.

Note
Some endpoint data types and functions correspond exactly to those in the X/Open Transport Interface (XTI), from which Open Transport derives its application programming interface. Appendix A lists these data types and functions. You can refer to these data types and functions by their Open Transport names or their corresponding XTI names. For example, you can refer to the Open Transport function `OTBind` by the XTI name `t_bind`. This chapter refers to endpoint data types and functions by their Open Transport names. ◆

Constants and Data Types

This section describes the constants and data types that you can use with endpoints. The data types include general types that you can use with any type of endpoint and specific types that you can use only with one type of endpoint. The general types (the `TEndpointInfo` structure, the `TBind` structure, and the `OTData` structure) are described first, just following the descriptions of constants.

OTData Constant

When transferring data that is noncontiguous, you need to use an `OTData` buffer instead of the `TNetbuf` structure. Open Transport provides a constant that you can use when you send or receive data to indicate that the value in the `TNetbuf` structure is actually a pointer to an `OTData` buffer.

```
enum {
    kNetbufDataIsOTData     = (size_t)0xfffffffeU
};
```

OTBuffer Constant

When receiving data without making a copy, you need to point to an `OTBuffer` pointer. Open Transport provides a constant that you can use instead of the `nbytes` parameter of the `OTRcv` function or the `udata.maxlen` field used with other receive functions to indicate that you are doing this.

```
enum {
    kNetbufDataIsOTBufferStar= (size_t)0xfffffffdU
};
```

Buffer Types Enumeration

Each of the structures described by the structure types enumeration contains fields that point to different kinds of buffers. When you allocate memory for such a structure using the `OTAlloc` function (page 3-102), you can also specify that these buffers be allocated by specifying one or more of the constant names given by the buffer types enumeration.

The length of the allocated buffer is at least as large as the size returned for the endpoint by the `OTGetEndpointInfo` function (page 3-92). For each buffer allocated, the `OTAlloc` function sets the `maxlen` field to the length of the buffer, and the `len` field to 0. To specify more than one constant name, use the `bitOR` operator to combine values.

```
enum {
        T_ADDR          = 0x01,
        T_OPT           = 0x02,
        T_UDATA         = 0x04,
        T_ALL           = 0xffff
};
```

Constant descriptions

`T_ADDR` The `addr` field of the `TBind`, `TCall`, `TUDErr`, `TUnitRequest`, 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 structure being allocated.
`T_ALL`	All relevant fields of the desired structure are allocated.

Endpoint Service Types

Open Transport uses the `servtype` field of the `TEndpointInfo` structure (page 3-58) to indicate the kind of service the endpoint provides. The constant names that Open Transport can return for this field are given by the endpoint service enumeration:

```
enum {
        T_COTS              = 1,
        T_COTS_ORD          = 2,
        T_CLTS              = 3,
        T_TRANS             = 5,
        T_TRANS_ORD         = 6,
        T_TRANS_CLTS        = 7
};
```

Constant descriptions

`T_COTS`	Connection-oriented transactionless service without orderly release.
`T_COTS_ORD`	Connection-oriented transactionless service with optional orderly release.
`T_CLTS`	Connectionless transactionless service.
`T_TRANS`	Connection-oriented transaction-based service without orderly release.
`T_TRANS_ORD`	Connection-oriented transaction-based service with optional orderly release.
`T_TRANS_CLTS`	Connectionless transaction-based service.

Open Transport Flags

Open Transport uses the `OTFlags` enumeration to specify additional information about data that is being transmitted. The constant names that Open Transport can return for this field are given by the Open Transport flags enumeration.

```
typedef UInt32      OTFlags;

enum {
    T_MORE              = 0x001,/* More data to come in message */
    T_EXPEDITED         = 0x002,/* Data is expedited, if possible */
    T_ACKNOWLEDGED      = 0x004,/* Acknowledge transaction */
    T_PARTIALDATA       = 0x008,/* Partial data - more coming */
    T_NORECEIPT         = 0x010/* No event on transaction done */
    T_TIMEDOUT          = 0x020/* Reply timed out */
};
```

Constant descriptions

`T_MORE`	There is more data for the current TSDU or ETSDU. The next send or receive operation will handle additional data for this TSDU or ETSDU.
`T_EXPEDITED`	The data is sent as expedited data if the endpoint supports expedited data.
`T_ACKNOWLEDGED`	The transaction must be acknowledged before the send or receive function can complete.
`T_PARTIALDATA`	There is more data for the current TSDU or ETSDU. Unlike `T_MORE`, `T_PARTIALDATA` does not guarantee that the next send or receive operation will handle additional data for this TSDU or ETSDU.
`T_NORECEIPT`	There is no need to send a `T_REPLY_COMPLETE` event to complete the transaction. If you don't need to know when the transaction is actually done, you can set this flag to improve performance.
`T_TIMEDOUT`	The reply timed out. If a protocol such as ATP loses the acknowledgement for a transaction that needs to be acknowledged, the transaction will eventually time out. Since the reply didn't really fail (it just timed out), Open Transport can send a `T_REPLY_COMPLETE` event to complete the transaction and set this flag to explain what happened.

Endpoint Flags

Open Transport uses the `flags` field of the `TEndpointInfo` structure (page 3-58) to specify additional information about the endpoint. The constant names that

Open Transport can return for this field are given by the endpoint flags enumeration:

```
enum {
        T_SENDZERO              = 0x001,
        T_XPG4_1                = 0x002,
        T_CAN_SUPPORT_MDATA     = 0x10000000,
        T_CAN_RESOLVE_ADDR      = 0x40000000,
        T_CAN_SUPPLY_MIB        = 0x20000000
};
```

Constant descriptions

`T_SENDZERO` This endpoint lets you send and receive zero-length TSDUs.

`T_XPG4_1` This endpoint supports the `OTGetProtAddress` function.

`T_CAN_SUPPORT_MDATA` This endpoint supports `M_DATA`, that is, it permits sending raw packets. This is Streams-specific and is found in the `mistreams.h` header file. When you send such a packet, set the packet's `addr.buf` field to a null value and set its `addr.len` field to –1. This indicates that the data portion of the `TUnitData` structure already has the header information in it.

`T_CAN_RESOLVE_ADDR` This endpoint supports the `OTResolveAddress` function.

`T_CAN_SUPPLY_MIB` This endpoint can supply the Management Information Base (MIB) data used by the Simple Network Management Protocol (SNMP).

Endpoint States

The `OTGetEndpointState` function (page 3-93) returns an integer specifying the current state of an endpoint. Integer values and their corresponding constant names are given by the endpoint states enumeration. For information about endpoint states, see the section "Endpoint States" on page 3-13.

```
enum {
        T_UNINIT        = 0,    /* endpoint is uninitialized        */
        T_UNBND         = 1,    /* endpoint is unbound          */
        T_IDLE          = 2,    /* idle  */
```

```
        T_OUTCON        = 3,    /* outgoing connection pending */
        T_INCON         = 4,    /* incoming connection pending */
        T_DATAXFER      = 5,    /* data transfer     */
        T_OUTREL        = 6,    /* outgoing orderly release */
        T_INREL         = 7     /* incoming orderly release */
};
```

Constant descriptions

`T_UNINIT`	This endpoint has been closed and destroyed.
`T_UNBND`	This endpoint is initialized but has not yet been bound to an address.
`T_IDLE`	This endpoint has been bound to an address and is ready for use: connectionless endpoints can send or receive data; connection-oriented endpoints can initiate or listen for a connection.
`T_OUTCON`	This endpoint has initiated a connection and is waiting for the peer endpoint to accept the connection.
`T_INCON`	This endpoint has received a connection request but has not yet accepted or rejected the request.
`T_DATAXFER`	This connection-oriented endpoint can now transfer data because the connection has been established.
`T_OUTREL`	This endpoint has issued an orderly disconnect that the peer has not acknowledged. The endpoint can continue to read data, but must not send any more data.
`T_INREL`	This endpoint has received a request for an orderly disconnect, which it has not yet acknowledged. The endpoint can continue to send data until it acknowledges the disconnection request, but it must not read data.

Structure Types

The `OTAlloc` function (page 3-102) allocates a data structure that you specify using one of the constant names given by the structure types enumeration:

```
enum {
        T_BIND              = (OTStructType)1,
        T_OPTMGMT           = (OTStructType)2,
        T_CALL              = (OTStructType)3,
        T_DIS               = (OTStructType)4,
```

```
        T_UNITDATA          = (OTStructType)5,
        T_UDERROR           = (OTStructType)6,
        T_INFO              = (OTStructType)7,
        T_REPLYDATA         = (OTStructType)8,
        T_REQUESTDATA       = (OTStructType)9,
        T_UNITREQUEST       = (OTStructType)10,
        T_UNITREPLY         = (OTStructType)11
};
```

Constant descriptions

`T_BIND`	Specifies the `TBind` structure (page 3-61).
`T_OPTMGMT`	Specifies the `TOptMgmt` structure, described in the chapter "Option Management" in this book.
`T_CALL`	Specifies the `TCall` structure (page 3-72).
`T_DIS`	Specifies the `TDiscon` structure (page 3-161).
`T_UNITDATA`	Specifies the `TUnitData` structure (page 3-65).
`T_UDERROR`	Specifies the `TUDError` structure (page 3-67).
`T_INFO`	Specifies the `TEndpointInfo` structure (page 3-58).
`T_REPLYDATA`	Specifies the `TReply` structure (page 3-77).
`T_REQUESTDATA`	Specifies the `TRequest` structure (page 3-76).
`T_UNITREQUEST`	Specifies the `TUnitRequest` structure(page 3-68).
`T_UNITREPLY`	Specifies the `TUnitReply` structure (page 3-70).

The TEndpointInfo Structure

The `TEndpointInfo` structure describes the initial characteristics of an endpoint that you opened by calling the `OTOpenEndpoint` function (page 3-92) or the `OTAsyncOpenEndpoint` function (page 3-86). These functions return as a parameter a pointer to a `TEndpointInfo` structure, if there is one. The `TEndpointInfo` structure is optional; some endpoints might not provide one, depending on which protocol modules they use. You can also obtain a pointer to the `TEndpointInfo` structure by calling the `OTGetEndpointInfo` function (page 3-92) or the `OTBind` function (page 3-87).

You use the `TEndpointInfo` structure to find out how large a buffer you must allocate to send or receive information for the endpoint and what kind of services the endpoint provides.

IMPORTANT

It is recommended that you do not hard-code `TEndpointInfo` field values into your application when specifying the maximum length of buffers because these values might change. ▲

The `TEndpointInfo` structure is defined by the `TEndpointInfo` data type.

```
struct TEndpointInfo
{
    SInt32      addr;           /* maximum size of an address */
    SInt32      options;        /* maximum size of options */
    SInt32      tsdu;           /* normal data transmit unit size*/
    SInt32      etsdu;          /* expedited data transmit unit size */
    SInt32      connect;        /* maximum data size on connect */
    SInt32      discon;         /* maximum data size on disconnect */
    UInt32      servtype;       /* service type */
    UInt32      flags;          /* flags     */
};
typedef struct TEndpointInfo TEndpointInfo;
```

Field descriptions

`addr` A value greater than 0 indicates the maximum size (in bytes) of a protocol address to which you can bind this endpoint. A value of `T_INVALID` indicates that this endpoint does not allow access to protocol addresses; it is being used for serial communication.

`options` A value greater than 0 indicates the maximum number of bytes needed to store the protocol-specific options that this endpoint supports, if any. A value of `T_INVALID` indicates that this endpoint has no protocol-specific options that you can set.

`tsdu` For a transactionless endpoint, a positive value indicates the maximum number of bytes in a transport service data unit (TSDU) for this endpoint. A value of `T_INFINITE` indicates that there is no limit to the size of a TSDU. A value of 0 indicates that the provider does not support the concept of a TSDU. This means that you can send a data stream with no logical boundaries preserved across a connection. A value of `T_INVALID` indicates that this

endpoint cannot transfer normal data (as opposed to expedited data).

For a transaction-based endpoint, this field indicates the maximum number of bytes in a response.

`etsdu` For a transactionless endpoint, a positive value indicates the maximum number of bytes in an expedited transport service data unit (ETSDU) for this endpoint. A value of `T_INFINITE` indicates that there is no limit to the size of a ETSDU. A value of 0 indicates that this endpoint does not support the concept of an ETSDU. This means that you can send an expedited data stream with no logical boundaries preserved across a connection. A value of `T_INVALID` indicates that this endpoint cannot transfer expedited data.

For a transaction-based endpoint, this field indicates the maximum number of bytes in a request.

`connect` For a connection-oriented endpoint, a value greater than 0 indicates the maximum amount of data (in bytes) that you can send with the `OTSnd` function (page 3-140) or `OTAccept` function (page 3-137). A value of `T_INVALID` indicates that this endpoint does not let you send data with these functions. This field is meaningless for other types of endpoints.

`discon` For a connection-oriented endpoint, a value greater than 0 indicates the maximum amount of data (in bytes) that you can send using the `OTSndDisconnect` function (page 3-159) and the `OTSndOrderlyDisconnect` function (page 3-163). A value of `T_INVALID` indicates that this endpoint does not let you send data with these functions. This field is meaningless for other types of endpoints.

`servtype` A constant that indicates what kind of service the endpoint provides. Possible values are given by the endpoint service enumeration (page 3-54).

`flags` A bit field that provides additional information about the endpoint. Possible values are given by the endpoint flags enumeration (page 3-55).

The TBind Structure

The `TBind` structure describes the protocol address to which an endpoint is currently bound or connected, or specifies the protocol address to which you wish to bind or connect the endpoint. For a connection-oriented endpoint, the TBind structure also specifies the actual or desired number of connection requests that can be concurrently outstanding for the endpoint.

You pass the `TBind` structure as a parameter to the `OTBind` function (page 3-87), the `OTGetProtAddress` function (page 3-96), and `OTResolveAddress` function (page 3-98).

The `TBind` structure is defined by the `TBind` data type.

```
struct TBind
{
    TNetbuf     addr;
    OTQLen      qlen;
};
typedef struct TBind            TBind;
```

Field descriptions

`addr` A `TNetbuf` structure that contains information about an address. The `addr.maxlen` field specifies the maximum size of the address, the `addr.len` field specifies the actual length of the address, and the `addr.buf` field points to the buffer containing the address.

When specifying an address, you must allocate a buffer for the address and initialize it, you must set the `addr.buf` field to point to this buffer, and you must set the `addr.len` field to the size of the address.

When requesting an address, you must allocate a buffer in which the address is to be placed, you must set the `addr.buf` field to point to this buffer, and you must set the `addr.maxlen` field to the maximum size of the address that is being returned. You determine this value by examining the `addr` field of the `TEndpointInfo` structure for the endpoint.

`qlen` For a connection-oriented endpoint, the maximum number of connection requests that can be concurrently outstanding for this endpoint. For more information, see

the description of the OTBind function (page 3-87). For connectionless endpoints, this field has no meaning.

The OTData Structure

You use the OTData structure to specify the location and size of noncontiguous data. You can use this structure in place of the normal TNetbuf structure to describe a data buffer when sending data using the OTSndUData function (page 3-111), the OTSndURequest function (page 3-117), the OTSndUReply function (page 3-122), the OTSnd function (page 3-140), the OTSndRequest function (page 3-147), and the OTSndReply function (page 3-151).

▲ **WARNING**
The OTData structure is an Apple extension to the XTI specification. Using it might cause your program not to work when ported to other XTI/STREAMS environments. ▲

When transferring data, you normally specify a pointer to a TNetbuf structure that specifies the location and size of the buffer containing the data. However, you cannot use a TNetbuf structure to describe data that is noncontiguous. Instead you must use an OTData structure to describe each separate chunk of data. When the function that sends the data executes, it is able to locate all the chunks of data, given a pointer to the OTData structure that describes the first chunk.

Using the OTData structure enables you to send data that is not contiguous, but the total size of the data fragments must not exceed the maximum size of data that the endpoint can send. The limits for normal and expedited data are specified in the tsdu and etsdu fields of the TEndpointInfo structure for the endpoint.

Each OTData structure specifies the location of a data fragment, the size of the fragment, and the location of the OTData structure that specifies the location and size of the next data fragment. The data information structure is defined by the OTData type.

```
struct OTData {
    void*       fNext;
    void*       fData;
```

```
    size_t      fLen;
};
typedef struct OTData OTData;
```

Field descriptions

`fNext`	A pointer to the `OTData` structure that describes the next data fragment. Specify a `NULL` pointer for the last data fragment.
`fData`	A pointer to the data fragment.
`fLen`	A long specifying the size of the fragment in bytes.

The No-Copy Receive Buffer Structure

You use the no-copy receive buffer structure to specify that you wish to receive data without copying it. You can point to this structure when receiving data with the `OTRcvUData` function (page 3-115), the `OTRcvURequest` function (page 3-120), the `OTRcvUReply` function (page 3-125), the `OTRcv` function (page 3-144), the `OTRcvRequest` function (page 3-149), and the `OTRcvReply` function (page 3-154).

Note
If you are familiar with Streams `mblk_t` data structures, you can see that the no-copy receive buffer structure is just a slight modification of the `mblk_t` structure. ◆

You can only use this buffer for data; you cannot use it for the address or options that may be associated with the incoming data. For example, in the case of an incoming `TUnitData` structure, you can only capture the `udata` portion, not the `addr` or `opt` fields.

▲ **WARNING**
Under no circumstance write to this data structure. It is read-only. If you write to it, you can crash the system. ▲

The no-copy receive buffer structure is defined by the `OTBuffer` data type.

```
struct OTBuffer
{
    void*       fLink;
    void*       fLink2;
    OTBuffer*   fNext;
```

```
    UInt8*      fData;
    size_t      fLen;
    void*       fSave;
    UInt8       fBand;
    UInt8       fType;
    UInt8       fPad1;
    UInt8       fFlags;
};

typedef struct OTBuffer OTBuffer;
```

Field descriptions

`fLink`	Reserved.
`fLink2`	Reserved.
`fNext`	A pointer to the next `OTBuffer` structure in the linked chain. By tracing the chain of `fNext` pointers, you can access all of the data associated with the message.
`fData`	A pointer to the data portion of this `OTBuffer` structure.
`fLen`	The length of data pointed to by the `fData` field.
`fSave`	Reserved.
`fBand`	The band used for the data transmission. It must be a value between 0 and 255.
`fType`	The type of the data (normally `M_DATA`, `M_PROTO`, or `M_PCPROTO`).
`fPad1`	Reserved.
`fFlags`	The flags associated with the data (`MSGMARK`, `MSGDELIM`).

IMPORTANT

Once you have copied the data out of the no-copy receive buffer, you need to call the `OTReleaseBuffer` function as quickly as possible to return the buffer to Open Transport. ▲

In many cases, for performance reasons, drivers pass their actual DMA buffers when they return data. 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 adversely affects 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 seems necessary to hold on to the buffer for any length of time, overall performance is better if you instead make a copy of the data and return the buffer to the system.

Buffer Information Structure

The buffer information structure is provided for your convenience in keeping track of where you last left off in an `OTBuffer` structure. Because the no-copy receive buffer structure (`OTBuffer`) is read-only, you may need to copy the data in sections as you progress through the no-copy receive buffer.

The buffer information structure is defined by the `OTBufferInfo` data type.

```
    struct OTBufferInfo
{
    OTBuffer*   fBuffer;
    size_t      fOffset;
    UInt8       fPad;
};

typedef struct OTBufferInfo OTBufferInfo;
```

Field descriptions

`fBuffer`	A pointer to the no-copy receive buffer.
`fOffset`	An offset indicating how much of the buffer to read.
`fPad`	Reserved.

The TUnitData Structure

You use the `TUnitData` structure to describe the data being sent with the `OTSndUData` function (page 3-111) and the data being read with the `OTRcvUData` function (page 3-115); you pass this structure as a parameter to each of these functions.

The `TUnitData` structure is defined by the `TUnitData` type.

```
struct TUnitData
    {   TNetbuf     addr;
        TNetbuf     opt;
```

```
        TNetbuf     udata;
    };
typedef struct TUnitData TUnitData;
```

Field descriptions

`addr` A `TNetbuf` structure that contains information about an address.

In the `udata` parameter to the `OTSndUData` function, this field specifies the location and size of the destination address. You must allocate a buffer to hold the address and initialize the `addr.buf` field to point to that buffer. You must set the `addr.len` field to the length of the address.

In the `udata` parameter to the `OTRcvUData` function, on return, this field specifies the location and size of the address of the endpoint that has sent the data. You must allocate a buffer to contain the address, initialize the `addr.buf` field to point to it, and set the `addr.maxlen` field to specify its maximum size.

`opt` A `TNetbuf` structure that contains information about options.

In the `udata` parameter to the `OTSndUData` function, this field specifies the location and size of options. You must allocate a buffer to hold the options and initialize the `opt.buf` field to point to that buffer. You must set the `opt.len` field to the length of the options buffer. If you do not want to specify any options, set the `opt.len` field to 0.

In the `udata` parameter to the `OTRcvUData` function, on return, this field contains any association-related options specified by the endpoint sending data. To read these options, you must allocate a buffer into which the provider can place the options; you must set the `opt.buf` field to point to the buffer; and you must set the `opt.maxlen` field to the maximum size of the buffer.

`udata` A `TNetbuf` structure that contains information about the data being transferred.

In the `udata` parameter to the `OTSndUData` function, this field specifies the location and size of the buffer containing the data to be sent. You must allocate a buffer for the data and initialize the `udata.buf` field to point to that buffer. You

must set the udata.len field to the size of the data being sent.

If you are sending data that is not stored contiguously, the udata.buf field is a pointer to an OTData structure that describes the first data fragment. In this case, you must set the udata.len field to the constant kNetbufDataIsOTData.

In the udata parameter to the OTRcvUData function, this field specifies the location and size of the buffer into which the data being received is going to be placed when the function returns. You must allocate a buffer for the data, set the udata.buf field to point to it, and set the udata.maxlen field to the maximum length of this buffer.

If you are doing a no-copy receive, the udata.buf field is a pointer to an OTBuffer pointer. In this case, you must set the udata.maxlen field to the constant kNetbufDataIsOTBufferStar.

The TUDErr Structure

The TUDErr structure points to information that explains why the OTSndUData function (page 3-111) has failed. You pass this structure as a parameter to the OTRcvUDErr function (page 3-113).

The TUDErr structure is defined by the TUDErr type.

```
struct TUDErr
    {   TNetbuf     addr;
        TNetbuf     opt;
        SInt32      error;
    };
typedef struct TUDErr TUDErr;
```

Field descriptions

addr — A TNetbuf structure that contains information about the destination address of the data sent using the OTSndUData function. The OTRcvUDErr function fills in this structure when the function returns. You must allocate a buffer to contain the address, initialize the addr.buf field to point to it, and set the addr.maxlen field to specify its maximum size.

opt	A TNetbuf structure that contains information about the options associated with the data sent using the OTSndUData function. The OTRcvUDErr function fills in this structure when the function returns. If you want to know this information, you must allocate a buffer to contain the option data, initialize the opt.buf field to point to it, and initialize the opt.maxlen field to specify the maximum size of the buffer. If you are not interested in option information, set the opt.len field to 0.
error	A long that, on return, specifies a protocol-dependent error code for the OTSndUData function that failed.

The TUnitRequest Structure

You use the TUnitRequest structure to specify information about the data being sent with the OTSndURequest function (page 3-117) and the data being read with the OTRcvURequest function (page 3-120); you pass a pointer to this structure as a parameter to each of these functions.

The TUnitRequest structure is defined by the TUnitRequest data type.

```
struct TUnitRequest
{       TNetbuf         addr;
        TNetbuf         opt;
        TNetbuf         udata;
        OTSequence      sequence;
    };
    typedef struct TUnitRequest TUnitRequest;
```

Field descriptions

addr	A TNetbuf structure that contains information about an address. In the req parameter to the OTSndURequest function, this field specifies the location and size of a buffer containing the address of the responder. You must allocate a buffer for the address and specify the address. You must set the addr.buf field to point to this buffer and set the addr.len field to the length of the address. In the req parameter to the OTRcvURequest function, this field specifies the location and size of a buffer containing

the address of the endpoint that made the request; the field is filled in by the `OTRcvURequest` function when it returns. You must allocate a buffer to hold address information and set the `addr.buf` field to point to it. You must also set the `addr.maxlen` field to the maximum size of the address.

`opt` A `TNetbuf` structure that contains information about the options associated with this request.

In the `req` parameter to the `OTSndURequest` function, this field specifies the location and size of a buffer containing the options you want to negotiate. You must allocate a buffer that contains the option information and set the `opt.buf` field to point to it. You must set the `opt.len` field to the length of the option data or to 0 if you don't want to specify any options.

In the `req` parameter to the `OTRcvURequest` function, this field specifies the location and size of a buffer containing the association-related options specified by the requester. Otherwise, this buffer is empty. When the `OTRcvURequest` function returns, it places option information in this buffer. You must allocate a buffer to contain the option information and set the `opt.buf` field to point to this buffer. You must set the `opt.maxlen` field to the maximum size necessary to hold option information for the endpoint.

`udata` A `TNetbuf` structure that contains information about the request data.

In the `req` parameter to the `OTSndURequest` function, this field specifies the location and size of a buffer containing the request data. You must allocate a buffer for the request data, initialize the `udata.buf` field to point to it, and set the `udata.len` field to the size of the request. The request size must not exceed the value for the `etsdu` field of the `TEndpointInfo` structure for the endpoint.

If you are sending data that is not stored contiguously, the `udata.buf` field is a pointer to an `OTData` structure that describes the first data fragment. In this case, you must set the `udata.len` field to the `kNetbufDataIsOTData` constant.

In the `req` parameter to the `OTRcvURequest` function, this field specifies the location and size of a buffer containing the request. You must allocate a buffer into which the

OTRcvURequest function can place the request and set the udata.buf field to point to it. You must set the udata.maxlen field to the maximum size of the request data.

If you are doing a no-copy receive, the udata.buf field is a pointer to an OTBuffer pointer. In this case, you must set the udata.maxlen field to the constant kNetbufDataIsOTBufferStar.

sequence — A long that specifies the transaction ID for this transaction.

You set this field to any desired value when you send the request.

When you read the request, this value is generated by the endpoint provider. You need to save this value and use it for the sequence field when sending a reply.

The TUnitReply Structure

You use the TUnitReply structure to specify the data being sent with the OTSndUReply function (page 3-122) and the data being read with the OTRcvUReply function (page 3-125). You pass a pointer to the TUnitReply structure as a parameter to each of these functions.

The TUnitReply structure is defined by the TUnitReply data type.

```
struct TUnitReply
    {
        TNetbuf         addr;
        TNetbuf         opt;
        TNetbuf         udata;
        OTSequence      sequence;
    };
    typedef struct TUnitReplyTUnitReply;
```

Field descriptions

addr — A TNetbuf structure that contains information about an address.

In the reply parameter to the OTSndUReply function, this field specifies the location and size of a buffer containing the address of the requester. You are not required to provide this information. If you do not want to provide

address information, set the `addr.len` field to 0. To specify an address, you must allocate a buffer for the address and initialize it to the destination address. Then you set the `addr.buf` field to point to the buffer and set the `addr.len` field to the length of the address.

In the `reply` parameter to the `OTRcvUReply` function, this field specifies the location and size of a buffer containing the address of the endpoint sending the reply. You must allocate a buffer into which the address is placed when the function returns, and you must set the `addr.buf` field to point to this buffer. You must also set the `addr.maxlen` field to the maximum size of the buffer.

`opt` A `TNetbuf` structure that contains information about the options associated with this reply.

In the `reply` parameter to the `OTSndUReply` function, this field specifies the location and size of a buffer containing the options that you set for this reply. You must set the `opt.len` field to the length of the options or to 0 if you don't want to specify any options.

In the `reply` parameter to the `OTRcvUReply` function, this field specifies the location and size of a buffer containing the association-related options that the responder has sent using the `OTSndUReply` function. You must allocate a buffer to hold option information and set the `reply.opt` field to point to it. When the `OTRcvUReply` function returns, it fills this buffer with option information. You must set the `reply.maxlen` field to the maximum size necessary to hold option information.

`udata` A `TNetbuf` structure that contains information about the reply data.

In the `reply` parameter to the `OTSndUReply` function, this field specifies the location and size of a buffer containing the reply data sent to the requester. You allocate a buffer that contains the reply data, set the `udata.buf` field to point to that buffer, and set the `udata.len` field to specify the size of the reply. The size cannot exceed the value specified for the `tsdu` field of the `TEndpointInfo` structure for the endpoint.

If you are sending data that is not stored contiguously, the `udata.buf` field is a pointer to an `OTData` structure that describes the first data fragment. In this case, you must set the `udata.len` field to `kNetbufDataIsOTData`.

In the `reply` parameter to the `OTRcvUReply` function, this field specifies the location and size of a buffer into which the function places the reply data on return. You must allocate a buffer to hold the data, set the `udata.buf` field to point to it, and set the `udata.maxlen` field to the maximum size of this buffer. The size must not exceed the value specified for the `tsdu` field of the `TEndpointInfo` structure for this endpoint.

If you are doing a no-copy receive, the `udata.buf` field is a pointer to an `OTBuffer` pointer. In this case, you must set the `udata.maxlen` field to the constant `kNetbufDataIsOTBufferStar`.

`sequence` A long that specifies the transaction ID for this transaction.

When sending a reply, you set this field to the value for this field that you read with the `OTRcvURequest` function.

When receiving a reply, if you have sent out multiple requests, you use this field to match incoming replies to outgoing requests.

The TCall Structure

You use the `TCall` structure to specify the options and data associated with establishing a connection. You pass a pointer to this structure as a parameter to the `OTConnect` function (page 3-131), the `OTRcvConnect` function (page 3-133), the `OTListen` function (page 3-135), and the `OTAccept` function (page 3-137).

The `TCall` structure is defined by the `TCall` data type.

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

```
};

typedef struct TCall TCall;
```

Field descriptions

`addr` A `TNetbuf` structure that specifies the location and size of a buffer containing an address. If you are using the `TCall` structure to send information, you must allocate a buffer and initialize it to contain the address, you must set the `addr.buf` field to point to the buffer, and you must set the `addr.len` field to the size of the address. If you are using the `TCall` structure to receive information, you must allocate a buffer into which the function can place the address when it returns, you must set the `addr.buf` field to point to this buffer, and you must set the `addr.maxlen` field to the maximum size of the address.

In the `sndCall` parameter to the `OTConnect` function, you must use this field to specify information about the address of the remote peer.

In the `rcvCall` parameter to the `OTConnect` function, on return, this field contains information about the address to which you are actually connected.

In the `call` parameter to the `OTRcvConnect` function, on return, this field contains information about the address to which you are actually connected.

In the `call` parameter to the `OTListen` function, on return, this field contains information about the address of the peer that requested the connection. The function returns the address in a format that you can use in future calls to the `OTConnect` function (page 3-131), the `OTSndDisconnect` function (page 3-159), the `OTSndOrderlyDisconnect` function (page 3-163), or the `OTAccept` function (page 3-137).

In the `call` parameter to the `OTAccept` function, you can use this field to specify information about the address of the peer that requested the connection. If you do not want to specify a value, set the `addr.len` field to 0.

In the `call` parameter to the `OTSndDisconnect` function, this field is reserved.

`opt` A `TNetbuf` structure that specifies the location and size of a buffer containing option information. If you are using the `TCall` structure to send information, you must allocate a buffer and initialize it to contain the option information, you must set the `opt.buf` field to point to the buffer, and you must set the `opt.len` field to the size of the option data. Set the `opt.len` field to 0 if you don't want to specify any options. If you are using the `TCall` structure to receive information, you must allocate a buffer into which the function can place option data when it returns, you must set the `opt.buf` field to point to this buffer, and you must set the `opt.maxlen` field to the maximum size of the option information.

In the `sndCall` parameter to the `OTConnect` function, you can use this field to specify the options you want to negotiate.

In the `rcvCall` parameter to the `OTConnect` function, on return, this field specifies the options that have been negotiated for this connection.

In the `call` parameter to the `OTRcvConnect` function, on return, this field specifies the options that have been negotiated for this connection.

In the `call` parameter to the `OTListen` function, on return, this field specifies the options that the peer has requested for this connection.

In the `call` parameter to the `OTAccept` function, you can use this field to specify the options that you want to use for the connection. Specifying 0 for the `opt.len` field means that you accept the connection unconditionally.

In the `call` parameter to the `OTSndDisconnect` function, this field is reserved.

`udata` A `TNetbuf` structure that specifies the location and size of a buffer containing data associated with a connection or disconnection request. Not all endpoints support the sending of data while establishing or tearing down a connection. Examine the `connect` or `discon` field of the `TEndpointInfo` structure for the endpoint to determine if the endpoint supports the sending of data and to find out the maximum size of the data you can send.

If you are using the `TCall` structure to send data, you must allocate a buffer and initialize it to contain the data, you must set the `udata.buf` field to point to the buffer, and you must set the `udata.len` field to the size of the data. If you are using the `TCall` structure to receive information, you must allocate a buffer into which the function can place the data when it returns, you must set the `udata.buf` field to point to this buffer, and you must set the `udata.maxlen` field to the maximum size of the data.

In the `sndCall` parameter to the `OTConnect` function, you can use this field to specify the data associated with the connection request.

In the `rcvCall` parameter to the `OTConnect` function, on return, this field specifies data that has been sent by the peer accepting the connection.

In the `call` parameter to the `OTListen` function, on return, this field specifies data that has been sent by the peer accepting the connection.

In the `call` parameter to the `OTAccept` function, you can use this field to specify data you want to send back to the peer that requested the connection.

In the `call` parameter to the `OTSndDisconnect` function, this field specifies the location and size of any data associated with the disconnection request.

`sequence` A long that is used by the `OTListen` and `OTAccept` functions to specify the connection ID.

In the `call` parameter to the `OTListen` function, on return, this field contains the connection ID of the incoming request.

In the `call` parameter to the `OTAccept` function, you must use this field to specify the connection ID of the connection request that you are accepting. This must be the same value that was passed to you by the `OTListen` function when you received the connection request.

In the `call` parameter to the `OTSndDisconnect` function, this field specifies the same connection ID as was returned by the `OTListen` function when the connection request was received. You must specify a value if you are calling the `OTSndDisconnect` function to reject a connection request.

This field is only meaningful if the endpoint is in the `T_INCON` state.

The TRequest Structure

You use the `TRequest` structure to specify the data being sent with the `OTSndRequest` function (page 3-147) and the data being read with the `OTRcvRequest` function (page 3-149). You pass a pointer to this structure as a parameter to each of these functions.

The `TRequest` structure is defined by the `TRequest` data type.

```
struct TRequest
    {
        TNetbuf         data;
        TNetbuf         opt;
        OTSequence      sequence;
    };
    typedef struct TRequest TRequest;
```

Field descriptions

`data` A `TNetbuf` structure specifying the location and size of the request data buffer.

In the `req` parameter to the `OTSndRequest` function, this field specifies the location and size of a buffer containing the request. You must allocate a buffer for the request data, set the `data.buf` field to point to it, and set the `data.len` field to the size of the request data. The size of the request cannot exceed the value specified for the `etsdu` field of the `TEndpointInfo` structure for the endpoint.

If you are sending data that is not stored contiguously, the `data.buf` field is a pointer to an `OTData` structure that describes the first data fragment. In this case, you must set the `udata.len` field to `kNetbufDataIsOTData`.

In the `req` parameter to the `OTRcvRequest` function, on return, this field specifies the location and size of a buffer containing the incoming request. You must allocate a buffer into which the request data is placed when the function returns and set the `data.buf` field to point to it. You must set the `data.maxlen` field to the maximum size of

the request data; this value cannot exceed the `etsdu` value specified for the endpoint. On return, the `data.len` field contains the actual length of the data sent.

If you are doing a no-copy receive, the `data.buf` field is a pointer to an `OTBuffer` pointer. In this case, you must set the `data.maxlen` field to the constant `kNetbufDataIsOTBufferStar`.

`opt` A `TNetbuf` structure specifying the location and size of the options buffer.

In the `req` parameter to the `OTSndRequest` function, this field specifies the location and size of a buffer containing the options you want to negotiate for this request. You must allocate a buffer that contains the option data, set the `opt.buf` field to point to it, and set the `opt.len` field to the size of the option data. Set the `opt.len` field to 0 if there are no options.

In the `req` parameter to the `OTRcvRequest` function, on return, this field specifies the location and size of a buffer containing the association-related options specified by the requester. You must allocate a buffer into which the endpoint provider can place the option data when the function returns, and set the `opt.buf` field to point to it. Set the `opt.maxlen` field to the maximum size of this buffer.

`sequence` A long that specifies the transaction ID of the current transaction.

You set this field to any desired value when you send the request.

When you read the request, this value is generated by the endpoint provider. You need to save this value and use it for the `sequence` field when sending a reply.

The TReply Structure

You use the `TReply` structure to specify the data being sent with the `OTSndReply` function (page 3-151) and the data being read with the `OTRcvReply` function (page 3-154). You pass this structure as a parameter to each of these functions.

The `TReply` structure is defined by the `TReply` data type.

```
struct TReply
    {
        TNetbuf     data;
        TNetbuf     opt;
        OTSequence sequence;
    };
    typedef struct TReply TReply;
```

Field descriptions

`data` A `TNetbuf` structure specifying the location and size of the reply buffer.

In the `reply` parameter to the `OTSndReply` function, this field specifies the location and size of a buffer containing the reply data. You must allocate and initialize a buffer that contains the data and set the `data.buf` field to point to it. You must set the `data.len` field to the size of the reply data. The size of the reply must not exceed the value specified for the `tsdu` field of the `TEndpointInfo` structure for this endpoint.

In the `reply` parameter to the `OTRcvReply` function, on return, this field specifies the size and location of a buffer into which the function places the data to be read. You must allocate a buffer for this data, set the `data.buf` field to point to it, and set the `data.maxlen` field to the maximum size of the buffer. This value must not exceed the value specified for the `tsdu` field of the `TEndpointInfo` structure for this endpoint.

If you are doing a no-copy receive, the `data.buf` field is a pointer to an `OTBuffer` pointer. In this case, you must set the `data.maxlen` field to the constant `kNetbufDataIsOTBufferStar`.

`opt` A `TNetbuf` structure describing the size and location of an option buffer.

In the `reply` parameter to the `OTSndReply` function, this field specifies the location and size of a buffer containing the options you want to set. You must allocate a buffer for the option values, set the `opt.buf` field to point to it, and set the `opt.len` field to the length of the options or to 0 if don't want to specify any options.

In the `reply` parameter to the `OTRcvReply` function, on return, this field specifies the location and size of a buffer containing the association-related options sent with the `OTSndReply` function. You must allocate a buffer for the option information, set the `opt.buf` field to point to it, and set the `opt.maxlen` field to the maximum size of the buffer.

`sequence` A long that specifies the transaction ID of the current transaction.

When sending a reply, you set this field to the value that you read with the `OTRcvURequest` function for this field.

When receiving a reply, if you have sent out multiple requests, you use this field to match incoming replies to outgoing requests.

The TDiscon Structure

You use the `TDiscon` structure to specify any user data sent with the disconnection and retrieved by the `OTRcvDisconnect` function (page 3-161). You pass this structure as a parameter to this function.

The `TDiscon` structure is defined by the `TDiscon` data type.

```
struct TDiscon
    {
        TNetbuf         udata;
        OTReason        reason;
        OTSequence      sequence;
    };
    typedef struct TDiscon TDiscon;
```

Field descriptions

`udata` A `TNetbuf` structure that is filled in with data sent with the `OTSndDisconnect` function. You must allocate a buffer in which the data is placed when the function returns, and you must initialize the `udata.maxlen` field to indicate the maximum size of the data that can be sent with the disconnection request.

`reason` A long specifying an error code that identifies the reason for the disconnection. These codes are supplied by the

	protocol. For additional information, consult the documentation provided for the protocol you are using.
`sequence`	A long specifying an outstanding connection request that has been rejected. This field is meaningful only when you have issued several connection requests to the same endpoint and are awaiting the results.

Functions

This section describes endpoint functions, provider functions that you use only with endpoints. The first four subsections—"Creating Endpoints," "Binding and Unbinding Endpoints," "Obtaining Information About an Endpoint," and "Allocating Structures" —describe functions that you can use with any endpoint. The remaining subsections describe functions that you can use only with specific types of endpoints, as indicated by the subsection title; for example, "Functions for Connectionless Transactionless Endpoints." Endpoint types are described in "Endpoint Types and Mode of Service" on page 3-7.

You can also use general provider functions with endpoints. General provider functions and structures are described in the reference section of the chapter "Providers" in this book.

Creating Endpoints

To transfer information, you need to create an endpoint and assign it an address. To create an endpoint, you call the `OTOpenEndpoint` or `OTAsyncOpenEndpoint` function. You must create an endpoint before calling any endpoint functions. After creating an endpoint, you must bind it by assigning it a protocol address. After binding, the endpoint is ready for use. When you finish using an endpoint, always call the function `OTCloseProvider` to close and delete the endpoint.

For more information about binding an endpoint, see "Binding and Unbinding Endpoints," beginning on page 3-86. For a description of the `OTCloseProvider` function, see the reference section of the chapter "Providers" in this book.

OTAsyncOpenEndpoint

Opens an endpoint and installs a notifier callback function for the endpoint.

The `OTAsyncOpenEndpoint` function is asynchronous, and creates an endpoint that operates asynchronously.

C INTERFACE

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

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`config`	A pointer to an endpoint configuration structure that specifies the endpoint's characteristics. You obtain a value for the `config` parameter by calling the `OTCreateConfiguration` function.
`oflag`	Reserved; must be set to 0.
`info`	A pointer to a `TEndpointInfo` structure to be filled in by the `OTAsyncOpenEndpoint` function. Specify `NULL` for this parameter if you do not want the `OTAsyncOpenEndpoint` function to return endpoint information.
`proc`	A pointer to a notifier callback function for this endpoint. If you do not provide a notifier function, your application cannot receive completion events, including the event advising you that the endpoint has been created. Specify `NULL` for this parameter if you do not want to provide a notifier function. In this case, you can use the `OTLook` function to poll for asynchronous events (but not for completion events).

`contextPtr` A pointer for your use. The endpoint provider passes this pointer value when calling the notifier function you specify in the `proc` parameter. You might use the `contextPtr` parameter, for example, to pass to your notifier function information about your application's current context.

DESCRIPTION

The `OTAsyncOpenEndpoint` function opens an endpoint having the characteristics specified by the `config` parameter. The `OTAsyncOpenEndpoint` function runs asynchronously, returning a result code as soon as the function has been queued for execution. How processing proceeds then depends on this result code.

If the result code is any except `kOTNoError`, an error occurred and Open Transport does not queue the function for execution. The `OTAsyncOpenEndpoint` function creates no endpoint and does not call the notifier function that you specified in the `proc` parameter.

If the result code is `kOTNoError`, the `OTAsyncOpenEndpoint` function attempts to create an endpoint. Then it calls the notifier function that you specified in the `proc` parameter, passing `T_OPENCOMPLETE` for the `code` parameter, a result code in the `result` parameter, and the endpoint reference for the newly created endpoint in the `cookie` parameter. It is recommended that you use the `OTAsyncOpenEndpoint` function to install a notifier function rather than using the `OTInstallNotifier` function to do it.

An endpoint created by the `OTAsyncOpenEndpoint` function operates in asynchronous mode, unless you change the endpoint's mode of execution by calling the `OTSetSynchronous` function. When an endpoint is in asynchronous mode, all provider functions that use the endpoint execute asynchronously.

By default, a newly created endpoint does not block and does not acknowledge sends. To change the endpoint's default mode of operation, you can call the `OTSetBlocking` function and the `OTIsAckingSends` function.

The preliminary state of an endpoint is `T_UNBND`, meaning that the endpoint is not bound to a protocol address. Before using the endpoint to transfer data, you must bind it to a protocol address by calling the `OTBind` function.

SPECIAL CONSIDERATIONS

The `OTAsyncOpenEndpoint` function destroys the configuration structure returned by the `OTCreateConfiguration` function. If you want to use the same configuration to open additional endpoints, you must obtain a valid copy of the configuration structure by calling the `OTCloneConfiguration` function before you call the `OTAsyncOpenEndpoint` function.

COMPLETION EVENT CODES

`T_OPENCOMPLETE`	0x20000007	The `OTAsyncOpenEndpoint` function has completed. The `cookie` parameter of the notifier function points to the endpoint reference for the new endpoint.

SEE ALSO

To create an `OTConfiguration` structure, use the `OTCreateConfiguration` function described in the chapter "Configuration Management" in this book.

To obtain a copy of the `OTConfiguration` structure, use the `OTCloneConfiguration` function described in the chapter "Configuration Management" in this book.

When you open an endpoint, Open Transport also creates the `TEndpointInfo` structure, which contains important information about the endpoint (page 3-58).

To create an endpoint synchronously, call the `OTOpenEndpoint` function (page 3-84).

You can use the `OTLook` function (page 3-95) to poll for asynchronous events.

Modes of execution are defined in the section "Modes of Operation," beginning on page 3-11. For information about changing an endpoint's mode of execution, see the chapter "Providers" in this book.

For information about notifier functions, see the chapter "Providers" in this book.

Endpoint states are defined and listed in"Endpoint States," beginning on page 3-13.

To close and delete an endpoint, call the `OTCloseProvider` function described in the chapter "Providers" in this book.

To bind a protocol address to an endpoint, call the `OTBind` function (page 3-87).

The `OTSetAsync`, `OTSetBlocking`, and `OTIsAckingSends` functions are described in the chapter "Providers" in this book.

OTOpenEndpoint

Opens an endpoint. This function is synchronous, and creates an endpoint that operates synchronously. It is strongly recommended that you use endpoints in asynchronous mode whenever possible.

C INTERFACE

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

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`config`	A pointer to an endpoint configuration structure that specifies the endpoint's characteristics. You obtain a value for the `config` parameter by calling the `OTCreateConfiguration` function. The `OTOpenEndpoint` function deletes the configuration structure when creating the endpoint or attempting to create it.
`oflag`	Reserved; must be set to 0.
`info`	A pointer to an `TEndpointInfo` structure to be filled in by the `OTOpenEndpoint` function. Specify `NULL` for this parameter if you do not want the `OTOpenEndpoint` function to provide endpoint information.
`err`	A pointer to the result code for this function.

DESCRIPTION

The `OTOpenEndpoint` function opens an endpoint having the configuration specified by the `config` parameter. The function returns an endpoint reference, by which you refer to the created endpoint when calling provider functions. If the `OTOpenEndpoint` function fails, its return value is `NULL`.

An endpoint created by the `OTOpenEndpoint` function operates in synchronous mode, unless you change the endpoint's mode of execution by calling the `OTSetAsynchronous` function. When an endpoint is in synchronous mode, all provider functions that use the endpoint execute synchronously.

By default, a newly created endpoint does not block and does not acknowledge sends. To change the endpoint's default mode of operation, you can call the `OTSetBlocking` function and the `OTIsAckingSends` function.

The initial state of an endpoint is `T_UNBND`, meaning that the endpoint is not bound to an address. Before using the endpoint to transfer data, you must bind it to an address by calling the `OTBind` function.

SPECIAL CONSIDERATIONS

The `OTOpenEndpoint` function changes the contents of memory and writes information to disk; your application should not call the `OTOpenEndpoint` function at interrupt time.

The `OTOpenEndpoint` function destroys the configuration structure returned by the `OTCreateConfiguration` function. If you want to use the same configuration to open additional endpoints, you must obtain a valid copy of the configuration structure before calling the `OTOpenEndpoint` function, by calling the `OTCloneConfiguration` function.

SEE ALSO

To create an `OTConfiguration` structure, use the `OTCreateConfiguration` function described in the chapter "Configuration Management" in this book.

To obtain a copy of the `OTConfiguration` structure, use the `OTCloneConfiguration` function described in the chapter "Configuration Management" in this book.

To create an endpoint that operates asynchronously, call the `OTAsyncOpenEndpoint` function (page 3-81).

Modes of execution are defined in the section "Modes of Operation," beginning on page 3-11. For information about changing an endpoint's mode of execution, see the chapter "Providers" in this book.

When you open an endpoint, Open Transport also creates the `TEndpointInfo` structure, which contains important information about the endpoint (page 3-58).

Endpoint states are defined and listed in"Endpoint States," beginning on page 3-13.

To close and delete an endpoint, call the `OTCloseProvider` function described in the chapter "Providers" in this book.

To bind a protocol address to an endpoint, call the `OTBind` function (page 3-87).

The `OTSetAsync`, `OTSetBlocking`, and `OTIsAckingSends` functions are described in the chapter "Providers" in this book.

Binding and Unbinding Endpoints

Binding an endpoint is the process of assigning an address to it. An address is the value by which a provider's highest-layer protocol module identifies the endpoint. For example, in AppleTalk, the protocol address of an ADSP endpoint is its network ID, node ID, and DDP socket number; in TCP/IP, the protocol address of a UDP endpoint is its port number and IP address. An endpoint must have a protocol address to transfer information.

You assign an address to an endpoint by calling the `OTBind` function. After binding, connectionless endpoints can send and receive data; connection-oriented endpoints can send and receive connection requests. If you use the `OTAccept` function (page 3-137) to pass off a connection request to another endpoint, it is not necessary to bind that endpoint first.

An endpoint can be bound to only one address at a time. If you no longer need to use an endpoint or if you want to change its address, you can unbind the endpoint using the `OTUnbind` function. In this case, Open Transport dissociates the endpoint from the address assigned to it. After the endpoint is unbound, you can close the endpoint using the `OTCloseProvider` function, or you can bind the endpoint to another address by using the `OTBind` function. You should not assume, after unbinding an endpoint, that you can bind the endpoint again to its former address. Of course, you can request the previous address when calling the `OTBind` function.

IMPORTANT
You must not close an endpoint during binding and unbinding; closing an endpoint deallocates memory reserved for it and the structures it uses. ▲

OTBind

Assigns an address to an endpoint.

C INTERFACE

```
OSStatus OTBind(EndpointRef ref, TBind* reqAddr, TBind* retAddr);
```

C++ INTERFACES

```
OSStatus TEndpoint::Bind(TBind* reqAddr, TBind* retAddr);
```

PARAMETERS

`ref` The endpoint reference of the endpoint that you are binding.

`reqAddr` A pointer to a `TBind` structure (page 3-61) that contains information about the address to which you want to bind the endpoint and the number of possible outstanding connection requests if this is a connection-oriented endpoint.

If you specify `NIL` for the `reqAddr` parameter, Open Transport chooses a protocol address for you and requests 0 as the endpoint's maximum number of concurrent outstanding connect indications.

If you want Open Transport to assign an address for you, set the `addr.len` field of the `TBind` structure to 0.

`retAddr` A pointer to a `TBind` structure (page 3-61) that, on return, indicates the address to which the endpoint is actually bound and, for connection-oriented endpoints, indicates the maximum

number of concurrent outstanding connect indications that this endpoint actually allows. The `TBind` structure is described on page 3-61.

You can set this parameter to `nil` if you do not care to know what address the endpoint is bound to or what the negotiated value of `qlen` is.

DESCRIPTION

You call the `OTBind` function to request an address that an endpoint be bound to. You can either use the `reqAddr` parameter to request that the endpoint be bound to a specific address or allow the endpoint provider to assign an address dynamically by passing `nil` for this parameter. Consult the documentation for the top-level protocol you are using to determine whether it is preferable to have the address assigned dynamically. The function returns the address to which the endpoint is actually bound in the `retAddr` parameter. This might be different from the address you requested, if you requested a specific address.

If you are binding a connection-oriented endpoint, you must use the `reqAddr->qlen` field to specify the number of connection requests that may be outstanding for this endpoint. The `retAddr->qlen` field specifies, on return, the actual number of connection requests allowed for the endpoint. This number might be smaller than the number you requested. Note that when the endpoint is actually connected, the number might be further decreased by negotiations taking place at that time.

If you call the `OTBind` function asynchronously and you have not installed a notifier function, the only way to determine when the function completes is to poll the endpoint using the `OTGetEndpointState` function. This function returns a `kOTStateChangeErr` until the bind completes. When the endpoint is bound, the state is either `T_UNBND` if the bind failed, or `T_IDLE` if it succeeded.

You can cancel an asynchronous bind that is still in progress by calling the `OTUnbind` function.

You must not bind more than one connectionless endpoint to a single address. Some connection-oriented protocols let you bind two or more endpoints to the same address. In such instances, you must use only one of the endpoints to listen for connection requests for that address. When binding the endpoint listening for a connection, you must set the `reqAddr->qlen` field of the `OTBind` function to a value greater than or equal to 1. When binding the other endpoints, you must set the `reqAddr->qlen` field to 0.

If you accept a connection for an endpoint that is also listening for connection requests, the address of that endpoint is deemed "busy" for the duration of the connection, and you must not bind another endpoint for listening to that same address. This requirement prevents more than one endpoint bound to the same address from accepting connection requests. If you have to bind another listening endpoint to the same address, you must first use the `OTUnbind` function to unbind the first endpoint or use the `OTCloseProvider` function to close it.

SPECIAL CONSIDERATIONS

In asynchronous mode, the endpoint provider might call your notifier function before the function's initial return.

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 the `OTAccept` function, if the responding address is to be the same as the called address.

COMPLETION EVENT CODES

`T_BINDCOMPLETE`	0x20000001	The `OTBind` function has completed. The `cookie` parameter passed to the notifier function points to the `retAddr` parameter.

VALID STATES

All except `T_UNINIT`

SEE ALSO

To unbind an endpoint call the `OTUnbind` function (described next).

The `TBind` structure (page 3-61) is used to specify the address to which the endpoint is bound.

You use the `OTCloseProvider` function, described in the chapter "Providers" in this book, to close a provider.

For additional information about binding multiple connection-oriented endpoints to the same address, see "Processing Multiple Connection Requests,"

beginning on page 3-33, the OTConnect function (page 3-131), and the OTAccept function (page 3-137).

For information on how to use this function with a TCP/IP protocol, see page 8-16 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-10 in the ADSP chapter, page 12-11 in the DDP chapter, and page 15-9 in the PAP chapter.

OTUnbind

Dissociates an endpoint from its address or cancels an asynchronous call to the OTBind function.

C INTERFACE

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

C++ INTERFACES

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

PARAMETERS

ref	The endpoint reference of the endpoint that you are unbinding.

DESCRIPTION

If you call the OTUnbind function asynchronously and you have not installed a notifier function, the only way to determine that the endpoint has been unbound is to use the OTGetEndpointState function to poll the state of the endpoint. The function returns the kOTStateChangeErr result when the OTUnbind function returns. If the function succeeds, the state of the endpoint is T_UNBND. If it fails, its state is T_IDLE.

After you unbind an endpoint, you can no longer use it to send or receive information. You can use the `OTCloseProvider` function to deallocate memory reserved for the endpoint, or you can use the `OTBind` function to associate it with another address and then resume transferring data or establishing a connection.

SPECIAL CONSIDERATIONS

In asynchronous mode, the endpoint provider might call your notifier function before the function's initial return.

VALID STATES

`T_IDLE`

COMPLETION EVENT CODES

`T_UNBINDCOMPLETE`	0x20000002	The `OTUnbind` function has completed. The `cookie` parameter of the endpoint's notifier function is not used.

SEE ALSO

To bind an endpoint, use the `OTBind` function, described on page 3-87.

To obtain information about the endpoint's state, use the `OTGetEndpointState` function, described on page 3-93.

The `OTCloseProvider` function is described in the chapter "Providers" in this book.

Obtaining Information About an Endpoint

You use the functions described in this section to obtain information about an endpoint. The `OTGetEndpointInfo` function returns information about the mode of service provided by the endpoint and the maximum size of the buffers used to specify address and option information and to hold data. Two functions return information about an endpoint's address: the `OTGetProtAddress` returns the endpoint's address and, if the endpoint is connected, the address of its peer. The `OTResolveAddress` function returns the protocol address that corresponds to an endpoint name. To obtain the state of the endpoint, you can call the

OTGetEndpointState function. To determine whether there are any asynchronous events pending for the endpoint, you can call the OTLook function. Finally, the OTSync function is provided to accommodate existing XTI applications that use this function.

In addition to the functions described in this section, you can use general provider functions to determine an endpoint's mode of execution (OTIsSynchronous) or an endpoint's mode of operation (OTIsAckingSends, OTIsNonBlocking). For more information about these functions, see the reference section of the chapter "Providers" in this book.

OTGetEndpointInfo

Obtains information about an endpoint that has been opened.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

ref	The endpoint reference of the endpoint whose characteristics you want to determine.
info	A pointer to a TEndpointInfo structure (page 3-58) that describes the endpoint's mode of service and the size of the buffers you can use to specify address and option information and to hold data.

DESCRIPTION

The OTGetEndpointInfo function returns information about

- the maximum size of buffers used to specify an endpoint's address and option values
- the maximum size of normal and expedited data you can transfer using this endpoint or, for transaction-based endpoints, the maximum size of requests and replies
- the size of data you can transfer when initiating or tearing down a connection
- the services supported by the endpoint
- any additional characteristics of this endpoint

COMPLETION EVENT CODES

`T_GETINFOCOMPLETE`	0x2000000A	`The OTGetEndpointInfo` function has completed. The `cookie` parameter of the endpoint's notifier function contains the `info` parameter.

VALID STATES

All

SEE ALSO

The `OTGetEndpointInfo` function returns a `TEndpointInfo` structure (page 3-58).

To obtain the current state of the endpoint, use the `OTGetEndpointState` function (described next).

OTGetEndpointState

Obtains the current state of an endpoint.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint whose state you want to determine.

DESCRIPTION

The `OTGetEndpointState` function returns an integer greater than or equal to 0 indicating the state of the specified endpoint. The endpoint state enumeration describes possible endpoint states and lists their decimal value.

If the function fails, it returns a negative integer specifying the error code. You must open an endpoint before you can determine its state.

You might need to know an endpoint's state in order to determine whether a function has completed or whether the endpoint is in an appropriate state for the function that you want to call next.

This function returns endpoint state information immediately, whether the endpoint is in synchronous or asynchronous mode.

VALID STATES

All

SEE ALSO

For general information about the services provided by an endpoint and the size of buffers it can use, use the `OTGetEndpointInfo` function (page 3-92).

The section "Endpoint States," beginning on page 3-13 explains how you use a knowledge of an endpoint's state to manage endpoints.

The endpoint state enumeration (page 3-56), describes possible endpoint states and lists their decimal value.

Use the `OTOpenEndpoint` function (page 3-84) or the `OTAsyncOpenEndpoint` function (page 3-86) to open an endpoint.

OTLook

Determines the current asynchronous event pending for an endpoint.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The endpoint reference of the endpoint.

DESCRIPTION

You use the `OTLook` function in one of two cases. First, if the endpoint is in synchronous mode, you can call the `OTLook` function to poll for incoming data or connection requests. Second, certain asynchronous events might cause a synchronous function to fail with the result `kOTLookErr`. For example, if you call `OTAccept` and the endpoint gets a `T_DISCONNECT` event, the `OTAccept` function returns with `kOTLookErr`. In this case, you need to call the `OTLook` function to determine what event caused the original function to fail. Table 3-7 on page 3-26 lists the functions that might return the `kOTLookErr` result and the events that can cause these functions to fail.

The `OTLook` function returns an integer value that specifies the asynchronous event pending for the endpoint specified by the `ref` parameter. On error, `OTLook` returns a negative integer corresponding to a result code.

If there are multiple events pending, the `OTLook` function first looks for one of the following events: `T_LISTEN`, `T_CONNECT`, `T_DISCONNECT`, `T_UDERR`, or `T_ORDREL`. If it finds more than one of these, it returns them to you in first-in, first-out order. After processing these events, the `OTLook` function looks for the `T_DATA`, `T_REQUEST`, and `T_REPLY` events. If it finds more than one of these, it returns them to you in first-in, first-out order. You cannot use the `OTLook` function to poll for completion events.

Unless you are operating exclusively in synchronous mode, it is recommended that you use notifier functions to get information about pending events for an endpoint.

VALID STATES

All

SEE ALSO

For additional information on asynchronous processing and on handling asynchronous and completion events, see the section "Handling Events for Endpoints," beginning on page 3-24 and the chapter "Providers" in this book.

Table 3-7 on page 3-26 lists the functions that might return the `kOTLookErr` result and the events that can cause these functions to fail.

The reference section of the chapter "Providers" in this book lists values returned for pending asynchronous events and describes their meanings.

For information on how to use this function with a TCP/IP protocol, see page 8-16 in the TCP/IP chapter.

OTGetProtAddress

Obtains the address to which an endpoint is bound and, if the endpoint is currently connected, obtains the address of its peer.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint whose local and peer address is sought.
`boundAddr`	A pointer to a `TBind` structure (page 3-61). The `boundAddr->addr` field is a `TNetBuf` structure that describes the address of the endpoint specified by the `ref` parameter. You must allocate a buffer for the address information and initialize the `addr.buf` field to point to that buffer. You must also initialize the `addr.maxlen` field to the maximum size of the address. If you are calling this function only to determine the address of the peer endpoint, you can set the `boundAddr` parameter to `NIL`. The `boundAddr->qlen` field is ignored.
`peerAddr`	A pointer to a `TBind` structure (page 3-61). If the endpoint specified by the `ref` parameter is currently connected, the `peerAddr->addr` field is a `TNetbuf` structure that describes the address of the endpoint's peer. The `boundAddr->qlen` field is ignored.

DESCRIPTION

The `OTGetProtAddress` function returns the address to which an endpoint is bound in the `boundAddr` parameter and, if the endpoint is currently connected, the address of its peer in the `peerAddr` parameter. Not all endpoints support this function. A value of `T_XPG4_1` in the `flags` field of the `TEndpointInfo` structure indicates that the endpoint does support this function.

You are responsible for initializing the buffers required to hold the local and peer addresses. The `addr` field of the `TEndpointInfo` structure specifies the maximum amount of memory needed to store the address of an endpoint. Use this value to set the size of the buffers.

The information returned by the `OTGetProtAddress` function is affected by the state of the endpoint specified by the `ref` parameter. If the endpoint is in the `T_UNBND` state, the `boundAddr->addr.len` field is set to 0. If the endpoint is not in the `T_DATAXFER` state, the `peerAddr->addr.len` field is set to 0.

If the endpoint is in asynchronous mode and a notifier is not installed, it is not possible to determine when the function completes.

COMPLETION EVENT CODES

`T_GETPROTADDRCOMPLETE`	0x20000008	The `OTGetProtAddress` function has completed. The `cookie` parameter of the notifier function contains the `peerAddr` parameter unless it is `nil`, in which case the `cookie` parameter contains the `boundAddr` parameter.

VALID STATES

All

SEE ALSO

The `TBind` structure (page 3-61) describes the address to which an endpoint is bound.

The `flags` field of the `TEndpointInfo` structure (page 3-55) indicates whether the endpoint supports this function.

For information on how to use this function with a TCP/IP protocol, see page 8-17 in the TCP/IP chapter.

For more information about the peer endpoint, see the description of the `OTAccept` function (page 3-137).

OTResolveAddress

Returns the protocol address that corresponds to the name of an endpoint.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint whose address is sought.
`req`	A pointer to a `TBind` structure (page 3-61). The `req->addr.buf` field points to a buffer containing the name of the endpoint, which must be in an appropriate format for the protocol family. For example, for AppleTalk this must be an NBP address.
`ret`	A pointer to a `TBind` structure (page 3-61). The `ret->addr.buf` field points to a buffer containing the lowest-level address that corresponds to the address pointed to by the `req->addr.buf` field of the `req` parameter.

DESCRIPTION

The `OTResolveAddress` function returns the lowest-level address for your endpoint. Not all endpoints support this function. A value of `CAN_RESOLVE_ADDR` in the `flags` field of the `TEndpointInfo` structure indicates that the endpoint does support this function. Using this function saves you the trouble of opening and closing a mapper provider if the only reason you have for opening the mapper is to look up the address corresponding to a specific endpoint name. You would still have to open the mapper if you needed to look up a name pattern—that is, if the name included any wildcard characters.

You are responsible for initializing the buffers described by the `req` and `ret` parameters required to hold the addresses. To determine how large these buffers should be, examine the `addr` field of the `TEndpointInfo` structure, which specifies the maximum amount of memory needed to store an address for the endpoint specified by the `ref` parameter.

If a notifier is not installed, it is not possible to determine when the `OTResolveAddress` function completes.

COMPLETION EVENT CODES

T_RESOLVEADDRCOMPLETE	0x20000009	The OTResolveAddress function has completed. The cookie parameter of the notifier function contains the result parameter.

VALID STATES

All states are valid except T_UNINT.

SEE ALSO

For additional information about the format used to describe the address passed in the ret parameter, please consult the documentation provided for the protocol you are using as the lowest-level protocol.

The TBind structure (page 3-61) describes the address to which an endpoint is bound.

The flags field (page 3-55) of the TEndpointInfo structure (page 3-58) indicates whether the endpoint supports this function.

OTSync

Ensures that the endpoint provider and the client have the same information about an endpoint's state.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint whose state information is being synchronized.

DESCRIPTION

The provider's and the client's knowledge about an endpoint's state might get out of sync if the provider and the client occupy different memory spaces. The current run-time environment does not support separate memory spaces; therefore, this function is currently provided so that existing XTI-based applications that make this call do not have to be modified.

If the `OTSync` function succeeds, it returns an integer value of 0 or greater that specifies the current state of the endpoint, as follows:

```
T_UNINIT        = 0
T_UNBND         = 1
T_IDLE          = 2
T_OUTCON        = 3
T_INCON         = 4
T_DATAXFER      = 5
T_OUTREL        = 6
T_INREL         = 7
```

If the `OTSync` function fails, it returns a negative integer corresponding to a result code.

If a notifier is not installed and the endpoint is in asynchronous mode, it is not possible to determine when the `OTSync` function completes.

COMPLETION EVENT CODES

`T_SYNCCOMPLETE`	0x2000000B	The `OTSync` function has completed. The `cookie` parameter of the notifier function is meaningless.

VALID STATES

All

SEE ALSO

If you are simply interested in obtaining information about the state of an endpoint, call the `OTGetEndpointState` function (page 3-93).

Allocating Structures

You use the `OTAlloc` and `OTFree` functions to allocate and free memory. These functions are mainly provided for XTI compatibility. In general, you should not use these functions to allocate and free structures on every call because this degrades performance. For a more detailed discussion of asynchronous processing and memory allocation, see the chapter "Providers" in this book.

OTAlloc

Allocates an XTI data structure.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The endpoint reference of the endpoint for which the data structure is allocated.

`structType` A long specifying the constant name of the structure for which memory is to be allocated. Possible values for the `structType` parameter are given by the structure types enumeration (page 3-57).

`fields` An integer specifying the structure fields for which buffers are to be allocated.

Each structure that you can specify for `structType`, except for `T_INFO`, contains at least one field of type `TNetbuf`. For each such field, you can use the `fields` parameter to specify that the buffer described by `TNetbuf` also be allocated. The length of the allocated buffer is at least as large as the size returned for the endpoint by the `OTGetEndpointInfo` function. For each buffer allocated, the `OTAlloc` function sets the `maxlen` field to the length of the buffer and sets the `len` field to 0.

You can specify one or more constant names for the `fields` parameter. Possible values for constant names are given by the buffer types enumeration (page 3-52). To specify more than one constant name, use the `bitOR` operator to combine values.

DESCRIPTION

The `OTAlloc` function allocates a data structure for use in a subsequent call. You use the `structType` parameter to specify the structure to be allocated and the `fields` parameter to specify the substructures to be allocated. If the `OTAlloc` function succeeds, it returns a pointer to the desired structure. The `OTAlloc` function is provided mainly for compatibility with XTI. Although using this function along with the `OTFree` function can save you coding work, this is at the price of slower performance. In general, you should not allocate and free structures on every call. Instead, you should declare structures that are to be passed as parameters to endpoint functions just as you would any other variables or data structures.

It is easiest to understand what the `OTAlloc` function does by considering what you would have to do if you did not use it. If you declared `structType` structures as normal data structures, you would have to declare the data structure and then initialize the `maxlen` and `buf` fields of every `TNetbuf` type field contained by the structure. To determine the appropriate size of each buffer, you would have to call the `OTGetEndpointInfo` function. For example, if you call the `OTGetProtAddress` function to get the protocol address of an endpoint, you must pass a parameter of type `TBind`. The `addr.buf` field of the `TBind` structure points to a buffer that is large enough to hold the endpoint's protocol address. To determine how large the buffer has to be, you call the `OTGetEndpointInfo` function; then you allocate the memory for the buffer and initialize the `addr.buf` field to point to the buffer and initialize the `addr.maxlen`

field to specify how large the buffer can be. The `OTAlloc` function does all this work for you. Given the previous example, if you make the call

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

the `OTAlloc` function allocates the `TBind` structure, initializes the `TNetbuf` field that is used to describe the endpoint address, and allocates memory for the buffer in which the address is to be stored. All buffers allocated are guaranteed to be of the appropriate size for the kind of endpoint specified by the `ref` parameter. You must not use the pointer returned by the `OTAlloc` function in calls to any other endpoint.

If the requested structure contains `TNetbuf` fields and you do not specify these fields using the `fields` parameter, the `OTAlloc` function sets the `maxlen`, `len`, and `buf` fields of these `TNetbuf` structures to 0.

SPECIAL CONSIDERATIONS

If you specify `T_UDATA` or `T_ALL` for the `fields` parameter and the endpoint information structure defines the `tsdu` or `etsdu` size for the endpoint to be of infinite length, the `OTAlloc` function does not allocate a data buffer for the endpoint.

VALID STATES

All

SEE ALSO

To deallocate memory allocated with the `OTAlloc` function, use the `OTFree` function (described next).

You use the structure types enumeration (page 3-57) to specify the structure for which memory is to be allocated.

You use the buffer types enumeration (page 3-52) to specify which `TNetbuf` structures should be allocated for the structure type you select.

The `TBind` structure (page 3-61) specifies the address of an endpoint.

The `TEndpointInfo` structure (page 3-58) specifies the maximum size of buffers used to hold an endpoint's address, options, and data.

To allocate raw memory, use the `OTAllocMem` function, and to deallocate the allocated raw memory, use the `OTFreeMem` function, both described in the chapter "Process Management."

OTFree

Frees memory allocated using the `OTAlloc` function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ptr`	A pointer to the structure to be deallocated. This is the pointer returned by the `OTAlloc` function.
`structType`	The name of the structure for which you allocated memory using the `OTAlloc` function. Possible constant names are given by the structure types enumeration. (page 3-57)

DESCRIPTION

In order to use the `OTFree` function, you must not have changed the memory allocated by the `OTAlloc` function for the structure specified by the `structType` parameter or for any of the buffers to which it points.

You are responsible for passing a `structType` parameter that exactly matches the type of structure being freed.

The `OTFree` function, along with the `OTAlloc` function, is provided mainly for compatibility with XTI.

VALID STATES

All

SEE ALSO

The `OTAlloc` function (page 3-102) allocates the memory `OTFree` deallocates.

You use one of the constant names given by the structure types enumeration (page 3-57) to specify the structure to be freed.

To allocate raw memory, use the `OTAllocMem` function, and to deallocate the allocated raw memory, use the `OTFreeMem` function, both described in the chapter "Process Management."

Checking a Buffer's Size

Open Transport provides a function, `OTCountDataBytes`, for checking a buffer's size before an endpoint handles it.

OTCountDataBytes

Returns the amount of data currently available to be handled.

C INTERFACE

```
OTResult OTCountDataBytes(EndpointRef ref, size_t* countPtr);
```

C++ INTERFACE

```
OTResult TEndpoint::CountDataBytes(size_t* countPtr);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint to which the data has been sent and which will be reading or otherwise using the data.

countPtr — A pointer to a buffer containing the size (in bytes) of the data in the topmost packet or stream buffer.

DESCRIPTION

If the function returns successfully, the countPtr parameter points to a buffer containing the the number of bytes in the message buffer at the top of the stream. How you might want to handle the data depends on which event is outstanding. For example, if you have a T_DATA outstanding, then the buffer indicates the data available to be read; if you have a T_DISCONNECT outstanding, then the buffer indicates the number of bytes in the disconnect.

Additionally, what the function counts depends on the type of endpoint. If it is packet-oriented, the function counts the number of bytes in the topmost packet; if it's stream-oriented, the function counts the total amount of nonexpedited data or the amount of expedited data in the topmost buffer at stream head. That is, if nonexpedited data was received in more than one piece, the function provides a count of the sum of the pieces, but if expedited data was received in multiple parts, the function only provides a count of the data in the topmost buffer.

If the buffer points at data to be read, this does not mean that this is all the data that was sent. You might need to do additional reads to get the rest of the data. You can call this function upon receipt of a T_DATA event to find out how much data is currently available and to determine whether you need to allocate larger buffers before calling a function that reads the data.

Because what this function counts depends on which event is the most current outstanding event and other events can occur before the function can complete, never use this count as more than a hint.

VALID STATES

All

Doing No-Copy Receives

Open Transport provides several functions for handling no-copy receives: the OTReleaseBuffer, OTReadBuffer, and the OTBufferDataSize functions.

OTReleaseBuffer

Returns the no-copy receive buffer to the system.

C INTERFACE

```
void OTReleaseBuffer(OTBuffer* buf);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`buf` A pointer to the no-copy receive buffer to be released.

DESCRIPTION

Once a no-copy receive is completed, you need to release the `OTBuffer` structure as quickly as possible by calling this function.

VALID STATES

All

SEE ALSO

The `OTBuffer` function (page 3-108) obtains the size of the no-copy receive buffer and the `OTReadBuffer` (page 3-109) function reads from this buffer.

The no-copy receive buffer structure is described by the `OTBuffer` data type (page 3-63).

OTBuffer

Obtains the size of the no-copy receive buffer.

C INTERFACE

```
size_t OTBuffer(OTBuffer* buf);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`buf`	A pointer to a no-copy receive buffer.

VALID STATES

All

SEE ALSO

The `OTReleaseBuffer` function (page 3-108) obtains the size of the no-copy receive buffer and the `OTReadBuffer` (page 3-109) function reads from this buffer.

The no-copy receive buffer structure is described by the `OTBuffer` data type (page 3-63).

OTReadBuffer

Reads the next portion of a no-copy receive buffer.

C INTERFACE

```
Boolean OTReadBuffer(OTBufferInfo* info, void* buf, size_t* len);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`info`	A pointer to the buffer information structure to be read.
`buf`	A pointer to a buffer into which to place the data.
`len`	The number of bytes actually read.

DESCRIPTION

This function returns `true` when it has read all of the bytes from the buffer information structure pointed to by the `info` parameter. It returns `false` when there are more bytes still to be read.

VALID STATES

All

SEE ALSO

The `OTReleaseBuffer` function (page 3-108) obtains the size of the no-copy receive buffer and the `OTBuffer` (page 3-108) function reads from this buffer.

The no-copy receive buffer structure is described by the `OTBuffer` data type (page 3-63).

Functions for Connectionless Transactionless Endpoints

You can use a connectionless transactionless endpoint to transfer data after the endpoint is bound and while it is in the `T_IDLE` state. Connectionless transactionless service used by protocols such as DDP, IP, PPP, or 802.2 is described at greater length in the section "Using Connectionless Transactionless Service," beginning on page 3-43. This section describes the functions used to send and receive data, `OTSndUData` and `OTRcvUData`. You use The `TUnitData` structure (page 3-65) with these functions to specify the data being transferred.

Some endpoint implementations do not detect an error in the attempt to send a datagram until after the `OTSndUData` function has returned successfully. In this case, Open Transport uses the `T_UDERR` event to notify the client sending the data. You can receive the event either by calling the `OTLook` function(page 3-95) or by including this case in your notifier function. To determine why the

OTSndUData function failed, you must call the OTRcvUDErr function, which is also described in this section. Due to the nature of connectionless transactionless service, you are not notified if the data fails to reach its destination.

OTSndUData

Sends data using a connectionless transactionless endpoint.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

ref	The endpoint reference of the endpoint sending the data.
udata	A pointer to a TUnitData structure (page 3-65) that specifies the data to be sent and its destination.

DESCRIPTION

If the endpoint is in synchronous, blocking mode, the OTSndUData function returns immediately. If flow-control restrictions prevent its sending the data, it retries the operation until it is able to send it. If the endpoint is in nonblocking mode, the OTSndUData function returns a kOTFlowErr result if flow-control restrictions prevent the data from being sent. When the endpoint provider is able to send the data, it calls your notifier function, passing T_GODATA for the code parameter. You can then call the OTSndUData function from your notifier to send the data. You can also retrieve this event by polling the endpoint using the OTLook function.

Some endpoint providers are not able to detect immediately whether you specified incorrect address or option information. In such cases, the provider

calls your notifier function when it detects the error, passing the `T_UDERR` for the `code` parameter to advise you that an error has occurred. You can determine the cause of this event by calling the `OTRcvUDErr` function and examining the value of the `uderr->error` parameter. It is important that you call the `OTRcvUDErr` function even if you are not interested in examining the cause of the error. Failing to do this leaves the endpoint in an invalid state for doing other sends and makes the endpoint provider unable to deallocate memory reserved for internal buffers associated with the send.

The next table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSndUData` function.

	Blocking	Nonblocking
Synchronous	The function returns when the provider lifts flow-control restrictions.	The function returns immediately.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately	The function returns immediately
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

SPECIAL CONSIDERATIONS

The `XTI_SLOWAT` option allows endpoints that support it to negotiate the minimum number of bytes that must have accumulated in the endpoint's internal send buffer before they are sent. If the endpoint you are using supports this option, you can negotiate a value using the `OTOptionManagement` function. Because you use the `OTOptionManagement` function to set this option, it affects all subsequent sends.

VALID STATES

`T_IDLE`

SEE ALSO

To read the data, the endpoint to which the data is sent uses the `OTRcvUData` function, (page 3-115).

You use the `TUnitData` structure (page 3-65) to specify the data to be sent and its destination.

You use the `OTData` structure (page 3-62) to transfer noncontiguous data.

You use the `OTOptionManagement` function, described in the reference section of the chapter "Option Management" in this book to negotiate a value for the `XTI_SNDLOWAT` option.

You use the `OTRcvUDErr` function (described next) to retrieve information about the cause of a `T_UDERR` event.

For information on how to use this function with a TCP/IP protocol, see page 8-18 in the TCP/IP chapter.

For information on how to use this function with the AppleTalk DDP protocol, see page 12-11 in the DDP chapter.

You use the `OTLook` function (page 3-95) to retrieve pending asynchronous events for an endpoint.

OTRcvUDErr

Clears an error condition indicated by a `T_UDERR` event and returns the reason for the error.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The endpoint reference of the endpoint that has attempted to send the data.

uderr	A pointer to a TUDErr structure (page 3-67) that specifies the reason for the error.

DESCRIPTION

You use the OTRcvUDErr function if you have called the OTSndUData function and the endpoint provider has issued the T_UDERR event to indicate that the send operation did not succeed. This usually happens when the endpoint provider cannot determine immediately that you have specified a bad address or option value. For example, assume that you are using AppleTalk and you specify an NBP address. If Open Transport cannot resolve the address, it sends a T_UDERR event to your notifier function. To clear the error condition and determine the cause of the failure, you must call the OTRcvUDErr function.

If the size of the option or error data returned exceeds the size of the allocated buffers, the OTRcvUDErr function returns with the result kOTBufferOverflowErr, but the error indication is cleared anyway.

If you do not need to identify the cause of the failure, you can set the uderr pointer to nil. In this case, the OTRcvUDErr function clears the error indication without reporting any information to you. It is important, nevertheless, that you actually call the OTRcvUDErr function to clear the error condition. If you don't call this function, the endpoint remains in an invalid state for doing other send operations, and the endpoint provider is unable to deallocate memory reserved for internal buffers associated with the send operation.

VALID STATES

T_IDLE

SEE ALSO

Open Transport uses the TUDErr structure (page 3-67) to return information about why the OTSndUData function (page 3-111) failed.

OTRcvUData

Reads data sent by a client using a connectionless transactionless protocol.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint receiving the data.
`udata`	A pointer to a `TUnitData` structure (page 3-65) that, on return, contains information about the data that has been received. See the description of the `TUnitData` structure for how to set this parameter when doing a no-copy receive.
`flags`	A pointer to an unsigned long variable whose bit setting, on return, indicates whether you need to retrieve more data. A value of `T_MORE` specifies that there is more data; a value of 0 specifies that there is no more data.

DESCRIPTION

When the `OTRcvUData` function returns, it passes a pointer to a `TUnitData` structure containing information about the data read and a pointer to a `flags` variable that is set to indicate whether there is more data to be retrieved. If the buffer pointed to by the `udata->udata.buf` field is not large enough to hold the current data unit, the endpoint provider fills the buffer and sets the `flags` parameter to `T_MORE` to indicate that you must call the `OTRcvUData` function again to receive additional data. Subsequent calls to the `OTRcvUData` function return 0 for the length of the address and option buffers until you receive the full data unit. The last unit to be received does not have the `T_MORE` flag set.

If the endpoint is in asynchronous mode or is not blocking and data is not available, the `OTRcvUData` function fails with the `kOTNoDataErr` result. The endpoint provider uses the `T_DATA` event to notify the endpoint when data becomes available. You can use a notifier function or the `OTLook` function to retrieve the event. Once you get the `T_DATA` event, you should continue calling the `OTRcvUData` function until it returns the `kOTNoDataErr` result.

It is possible that the provider generates an erroneous `T_DATA` event. This is the case when the provider calls your notifier, passing `T_DATA` for the `code` parameter; but when you execute the `OTRcvUData` function, it returns with a `kOTNoDataErr` result. If this happens, you should continue normal processing and assume that the next `T_DATA` event is genuine.

SPECIAL CONSIDERATIONS

The `XTI_RCVLOWAT` option allows endpoints that support it to negotiate the minimum number of bytes that must have accumulated in the endpoint's internal receive buffer before the endpoint provider generates a `T_DATA` event. If the endpoint you are using supports this option, you can negotiate a value using the `OTOptionManagement` function. Because you use the `OTOptionManagement` function to set this option, it affects all subsequent sends.

VALID STATES

`T_IDLE`

SEE ALSO

You can use the `OTLook` function (page 3-95) to retrieve pending asynchronous events for this endpoint.

For a description of the `OTOptionManagement` function, see the chapter "Option Management" in this book.

For information on how to use this function with the AppleTalk DDP protocol, see page 12-11 in the DDP chapter.

You use the `TUnitData` structure (page 3-65) to specify the size and location of buffers that contain information about the data that has been received.

Functions for Connectionless Transaction-Based Endpoints

You can use a connectionless transaction-based endpoint to transfer data after the endpoint is bound and while it is in the `T_IDLE` state. Connectionless transaction-based service used by protocols such as ATP is described at greater length in the section "Using Connectionless Transaction-Based Service," beginning on page 3-48.

This section describes the routines used to send and retrieve requests and replies: `OTSndURequest`, `OTRcvURequest`, `OTSndUReply`, and `OTRcvUReply`. This section also describes the `OTCancelURequest` function, which you use to cancel an outgoing request, and the `OTCancelUReply` function, which you use to cancel an incoming request.

OTSndURequest

Initiates a connectionless transaction by sending a request to the responder.

C INTERFACE

```
OSStatus OTSndURequest(EndpointRef ref, TUnitRequest* req,
                       int reqFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndURequest(TUnitRequest* req, int reqFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint making the request.

`req` A pointer to a `TUnitRequest` structure (page 3-68) that specifies the address of the responder, the request data, and the ID of this transaction.

`reqFlags`	A bitmapped long specifying whether delivery is guaranteed for both the requester and the responder (`T_ACKNOWLEDGED`) and whether you are sending the request data using additional calls to the `OTSndURequest` function (`T_MORE`). Use the `bitAND` operator to set both values.

DESCRIPTION

You use the `OTSndURequest` function to initiate a transaction. When the responder replies to your request, you use the `OTRcvUReply` function to read the reply. By default, the endpoint provider guarantees delivery for you, but not for the responder. That is, you will always find out whether your request was received, but the responder only receives acknowledgment that you received the reply if you have set the `T_ACKNOWLEDGED` flag in the `reqFlags` parameter when you send the request. Not all protocols honor this flag.

If the responder is an Open Transport endpoint, its provider generates a `T_REPLYCOMPLETE` event when you have read the reply. This happens whether or not the `T_ACKNOWLEDGED` flag is set, but if it is set, this guarantees that the reply was delivered. If you don't set this flag, the responder's call to the `OTSndUReply` function returns right away, and the responding endpoint receives no additional information as to whether the reply was received and the data was read.

Setting the `T_MORE` flag tells the endpoint provider that you are using several calls to the `OTSndURequest` function to send the request data. Note that even though you are using several calls, the request data, all put together, must still not exceed the value specified for the `etsdu` field in the endpoint's `TEndpointInfo` structure.

If the endpoint is in blocking mode and flow-control restrictions prevent the endpoint provider from accepting the `OTSndURequest` function, the provider waits to send the request until flow-control restrictions are lifted.

If the endpoint is in nonblocking mode and flow-control restrictions prevent the endpoint provider from accepting the `OTSndURequest` function, the function returns the `kOTFlowErr` result. When flow-control restrictions are lifted, the endpoint provider issues a `T_GODATA` event, which you can retrieve by polling the endpoint using the `OTLook` function or using a notifier function. When you get this event, you can retry sending the request.

The following table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSndURequest` function.

	Blocking	Nonblocking
Synchronous	The function returns when the provider lifts flow-control restrictions and the request has been sent to the protocol.	The function returns if flow-control restrictions are in effect or the request data has been sent to the protocol.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately	The function returns immediately.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

VALID STATES

`T_IDLE`

SEE ALSO

To determine the maximum size of the request data, you must call the `OTGetEndpointInfo` function (page 3-92) and examine the `etsdu` field of the `TEndpointInfo` structure that it returns.

You use the `TUnitRequest` structure (page 3-68) to specify the address of the responder, the request data, and the ID of this transaction.

You use the `OTData` structure (page 3-62) to transfer noncontiguous data.

To read the reply to an outgoing request, you must use the `OTRcvUReply` function (page 3-125).

For information on how to use this function with the AppleTalk ATP protocol, see page 14-10 in the ATP chapter.

You can poll for the `T_GODATA` event by calling the `OTLook` function (page 3-95).

OTRcvURequest

Reads a request sent by a client using a connectionless transaction-based protocol.

C INTERFACE

```
OSStatus OTRcvURequest(EndpointRef ref, TUnitRequest* req,
                       OTFlags* flags);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvURequest(TUnitRequest* req, OTFlags* flags);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint accepting the request.
`req`	A pointer to a `TUnitRequest` structure (page 3-68) that contains information about the request being received. See the description of the `TUnitRequest` structure for how to set this parameter when doing a no-copy receive.
`flags`	A long bitmapped field set by the endpoint provider that specifies whether the request is acknowledged (`T_ACKNOWLEDGED`) and whether there is more request data coming (`T_MORE`) or `(T_PARTIALDATA)`. A value of `T_MORE` indicates that the buffer you have allocated is too small to contain the reply. A value of `T_PARTIALDATA` indicates that the data unit being read does not contain the complete reply. It is possible that all flags are set.

DESCRIPTION

You use the `OTRcvURequest` function to read an incoming request. When the function returns, it fills in the `TUnitRequest` structure (referenced by the `req` parameter) with the address of the sender, the request data, and any association-related options pertaining to this request.

If the endpoint is in synchronous mode and is blocking, the `OTRcvURequest` function waits for a request to arrive. If the endpoint is in asynchronous mode or is not blocking, the `OTRcvURequest` function retrieves the next pending unread request or returns the `kOTNoDataErr` result if there are no pending requests.

If the endpoint is in asynchronous mode, the endpoint provider generates a `T_REQUEST` event when a request arrives. You can poll the endpoint using the `OTLook` function or use a notifier function to retrieve this event.

If the `T_MORE` bit is set in the `flags` parameter, this means your buffer is not large enough to hold the entire request. You must call the `OTRcvURequest` function again to retrieve more request data. Open Transport ignores the `addr` and `opt` fields of the `req` parameter for subsequent calls to the `OTRcvURequest` function. The `T_MORE` flag is not set for the last request packet to let you know that this is the last packet.

If the `T_PARTIALDATA` bit is set in the `flags` parameter, this means that the data you are about to read with the `OTRcvURequest` function does not constitute the entire request and that you must call the function again to read more of or the rest of the request.

If the `T_MORE` and the `T_PARTIALDATA` bits are both set, this means that the data you are about to read constitutes only part of the request and that your buffer is too small to contain even this chunk. In this case, you must call the function again until the `T_MORE` flag is clear. The `T_PARTIALDATA` bit is set only on the first call to the function.

If you are communicating with multiple requesters and the `OTRcvURequest` function returns with the `T_PARTIALDATA` flag set, it is possible that your next call to the `OTRcvURequest` function might not read the rest of the request because the next data unit coming in belongs to a different request. One way to handle this situation is to use the next call to the `OTRcvURequest` function to determine the sequence number of the incoming request (by setting `req->udata.len` to 0) and then, having determined which request data is coming in, read the data into the appropriate buffer.

The provider sets the `T_ACKNOWLEDGED` flag if the requester has set this flag when calling the `OTSndURequest` function. When this flag is set and you call the `OTSndUReply` function, Open Transport guarantees that your reply is acknowledged by the requester. This flag is set only on the first call to the `OTRcvURequest` function for any given transaction.

VALID STATES

T_IDLE

SEE ALSO

To determine the maximum size of the request data, you must call the OTGetEndpointInfo function (page 3-92) and examine the etsdu field of the TEndpointInfo structure (page 3-58) that it returns.

You use the TUnitRequest structure (page 3-68) to store information about the request being received.

You can poll for the T_REQUEST event by calling the OTLook function (page 3-95).

For information on how to use this function with the AppleTalk ATP protocol, see page 14-10 in the ATP chapter.

To respond to a request, you use the OTSndUReply function (described next).

OTSndUReply

Replies to a request sent by a client using a connectionless transaction-based protocol.

C INTERFACE

```
OSStatus OTSndUReply(EndpointRef ref, TUnitReply* reply,
                     OTFlags flags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndUReply(TUnitReply* reply, OTFlags flags);
```

PARAMETERS

ref The endpoint reference of the endpoint sending the reply.

reply	A pointer to a TUnitReply structure (page 3-70) that specifies the ID of this transaction and the reply data.
reqFlags	A bitmapped long, which you can set to T_MORE to indicate that you are sending more reply data with a subsequent call to the OTSndUReply function.

DESCRIPTION

You use the OTSndUReply function to send a reply. The TUnitReply structure that you pass in the reply parameter specifies the address of the requester, the reply data, and any options you want to specify for this reply. If you do not specify the requester's address, the endpoint provider uses the transaction ID value stored in the sequence field of the reply parameter to match the reply against a pending request and knows in this way where to send the request.

If requests are acknowledged and the provider is not able to send the reply, the function returns with the kETimedOutErr result. If requests are not acknowledged, the function returns immediately, and you have no way of knowing whether the reply was received and read.

If requests are not acknowledged, the provider generates a T_REPLYCOMPLETE event for asynchronous responders even if the requester has not acknowledged receipt of the reply. Thus, the only way for you to know whether this event actually means that the reply was received, is to examine the reqFlags field of the req parameter for the OTRcvURequest function. If the T_ACKNOWLEDGED flag is set, then the T_REPLYCOMPLETE event indicates that your reply was received. The cookie parameter passed to the notifier to indicate completion is set to the reply parameter.

The following table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSndUReply` function.

	Blocking	Nonblocking
Synchronous	The function returns when the provider lifts flow-control restrictions and the reply has been acknowledged or timed out (if the matching request was an acknowledged request).	For unacknowledged requests, the function returns immediately; for acknowledged requests, it returns when the reply has been acknowledged or time out.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately.	The function returns immediately.
	The provider calls your notifier, passing `T_REPLYCOMPLETE` for the `code` parameter when the reply is acknowledged or timed out.	The provider calls your notifier, passing `T_REPLYCOMPLETE` for the `code` parameter when the reply is acknowledged or timed out.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

COMPLETION EVENT CODES

`T_REPLYCOMPLETE`	0x20000004	The `OTSndUReply` function has completed. The `cookie` parameter of the notifier function points to the `reply` parameter.

VALID STATES

`T_IDLE`

SEE ALSO

To determine the maximum size of the reply data, you must call the `OTGetEndpointInfo` function (page 3-92) and examine the `tsdu` field of the `TEndpointInfo` structure that it returns.

You use the `TUnitReply` structure (page 3-70) to specify the ID of this transaction and the reply data.

You use the `OTData` structure (page 3-62) to describe noncontiguous data.

You use the `OTCancelUReply` function (page 3-129) to cancel an incoming request.

For information on how to use this function with the AppleTalk ATP protocol, see page 14-10 in the ATP chapter.

You examine the `reqFlags` field of the `req` parameter for the `OTRcvURequest` function (page 3-120) to determine whether the `T_REPLYCOMPLETE` event means that the reply was actually received.

OTRcvUReply

Reads a reply to a request sent by a client using a connectionless transaction-based protocol.

C INTERFACE

```
OSStatus OTRcvUReply(EndpointRef ref, TUnitReply* reply,
                     OTFlags* flagPtr);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvUReply(TUnitReply* reply, OTFlags* flags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint accepting the reply.

reply	A pointer to a TUnitReply structure (page 3-70) that contains information about the reply data and the ID of the transaction. See the description of the TUnitReply structure for how to set this parameter when doing a no-copy receive.
flagPtr	A pointer to a bitmapped long that is filled in by the endpoint provider to indicate whether there is more reply data to be read, in which case you must call the OTRcvUReply function again. A value of T_MORE indicates that the buffer pointed to by udata.buf is too small to contain the reply. A value of T_PARTIALDATA indicates that the data unit being read does not contain the complete reply. It is possible that both flags are set.

DESCRIPTION

You use the OTRcvUReply function to read the reply to a request that you have issued using the OTSndURequest function. The reply parameter points to buffers in which the function stores the reply, the address of the responder, any options connected with this transaction, and the transaction ID for this transaction.

If the endpoint is in asynchronous mode, the provider generates a T_REPLY event to let you know that reply data has arrived. If it should happen that the reply data is sent using multiple calls to the sending function, Open Transport does not generate additional T_REPLY events. To guard against this possibility, your notifier function should call the OTRcvUReply function until it returns the kOTNoDataErr result.

If a transaction has timed out awaiting reply data, the OTRcvUReply function returns a kETIMEDOUTErr result; the sequence field of the reply parameter specifies which request has timed out.

If you have issued multiple requests, it is not possible to know ahead of time how incoming replies match your requests. You must be prepared to receive a reply to any outstanding request. One way to manage this situation is to call the OTRcvUReply function with the reply->udata.maxlen field set to 0. The rest of the information returned by the function on this first call lets you know the sequence number of the reply as well as the flagPtr setting. Once you determine the matching request and the appropriate reply buffer, you can call the OTRcvUReply function a second time to read the actual reply data. On the second and subsequent reads, Open Transport sets the reply->opt.len field to 0. It is guaranteed that once a reply has been partially read, subsequent calls to OTRcvUReply read from that same reply until all the data has been read.

If the T_MORE bit is set in the flags parameter, this means your buffer is not large enough to hold the entire reply. You must call the OTRcvURequest function again to retrieve more request data. Open Transport ignores the addr and opt fields of the reply parameter for subsequent calls to the function. The T_MORE flag is not set for the last reply packet to let you know that this is the last packet.

If the T_PARTIALDATA bit is set in the flags parameter, this means that the data you are about to read with the OTRcvUReply function does not constitute the entire reply and that you must call the function again to read more of or the rest of the reply.

If the T_MORE and the T_PARTIALDATA bits are both set, this means that the data you are about to read constitutes only part of the reply and that your buffer is too small to contain even this chunk. In this case, you must call the function again until the T_MORE flag is clear. The T_PARTIALDATA bit is set only on the first call to the function.

If you are communicating with multiple responders and if the OTRcvUReply function returns with the T_PARTIALDATA flag set, it is possible that your next call to the function might not read the rest of the reply because the next data unit coming in belongs to a different reply. One way to handle this situation is to use the next call to the OTRcvReply function to determine the sequence number of the incoming reply (by setting req->udata.maxlen to 0) and then, having determined which reply data is coming in, read the data into the appropriate buffer.

VALID STATES

T_IDLE

SEE ALSO

You use the OTSndURequest function (page 3-117) to send a request.

For information on how to use this function with the AppleTalk ATP protocol, see page 14-10 in the ATP chapter.

You use the TUnitReply structure (page 3-70) to store information about the reply data and the ID of the transaction.

OTCancelURequest

Cancels a request that was made using the `OTSndURequest` function.

C INTERFACE

```
OSStatus OTCancelURequest(EndpointRef ref, OTSequence seq);
```

C++ INTERFACE

```
OSStatus Tendpoint::CancelURequest(OTSequence seq);
```

PARAMETERS

`ref` The endpoint reference of the endpoint that has sent the request being cancelled.

`seq` A long, specifying the transaction ID of the request you want to cancel. This is the same value as the one you specified for the `sequence` field of the `req` parameter when you called the `OTSndURequest` function.

If you specify 0 for this parameter, Open Transport cancels all outstanding requests. If you specify an invalid sequence number, Open Transport does not do anything.

DESCRIPTION

The `OTCancelURequest` function cancels the outgoing request whose transaction ID is specified by the `seq` parameter.

When you call the `OTSndURequest` function, the provider allocates memory for internal buffers for the transaction. Calling the `OTCancelURequest` function tells the endpoint provider that you are no longer interested in the transaction and that it can free up any memory or internal buffers associated with the transaction request identified by the `seq` parameter.

If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment. It is your responsibility to

deallocate memory that you have reserved for the address, options, and data buffers associated with the cancelled `OTSndURequest` function.

Use the `OTCancelURequest` function to cancel an *outgoing* request; use `OTCancelUReply` to cancel an *incoming* request.

VALID STATES

`T_IDLE`

SEE ALSO

You use the `OTSndURequest` function (page 3-117) to send a request.

You use the `OTCancelUReply` function (described next) to cancel an incoming request.

OTCancelUReply

Cancels a request that you have read using the `OTRcvURequest` function.

C INTERFACE

```
OSStatus OTCancelUReply(EndpointRef ref, OTSequence seq);
```

C++ INTERFACE

```
OSStatus TEndpoint::CancelUReply(OTSequence seq);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint that has sent the request being canceled.

`seq`	A long, specifying the transaction ID of the request being cancelled. Specify the same value as that value is passed in the `req` parameter to the `OTRcvURequest` function that you used to read this request. If you specify 0 for this parameter, Open Transport cancels all outstanding incoming requests. If you specify an invalid sequence number, Open Transport does not do anything.

DESCRIPTION

The `OTCancelUReply` function cancels the request whose transaction ID is specified by the `seq` parameter.

When you call the `OTRcvURequest` function, the endpoint provider allocates memory for internal buffers and assigns a sequence value to identify this transaction. Calling the `OTCancelUReply` function tells the provider that you are no longer interested in the transaction and that it can free up the memory and the sequence number associated with the cancelled transaction.

If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment. It is your responsibility to deallocate memory that you have reserved for the address, options, and data buffers associated with the cancelled `OTRcvURequest` function.

Use the `OTCancelUReply` function to cancel an *incoming* request; use the `OTCancelURequest` function to cancel an *outgoing* request.

VALID STATES

`T_IDLE`

SEE ALSO

You use the `OTRcvURequest` function (page 3-120) to read an incoming request.

You use the `OTCancelURequest` (page 3-128), to cancel an outgoing request.

Establishing A Connection

To use a connection-oriented endpoint, you must use the `OTBind` function to specify the number of outstanding connections that the listening endpoint

supports. Then you must use the functions described in this section to establish the connection. The endpoint initiating the connection uses the `OTConnect` and `OTRcvConnect` functions; the endpoint accepting the connection uses the `OTListen` and `OTAccept` functions. You use the same functions to establish a connection for both transactionless and transaction-based endpoints.

Once you have established a connection, you can send and receive data. How you do this depends on whether you are using transactionless or transaction-based service. After you are done transferring data and no longer need to stay connected, you must explicitly tear down the connection by using the functions described in the section "Tearing Down a Connection" on page 3-159.

OTConnect

Requests a connection to a remote peer.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint initiating the connection.
`sndCall`	A pointer to a `TCall` structure (page 3-72) that specifies the address of the remote peer, any data transmitted when establishing a connection, and any options for this connection.
`rcvCall`	A pointer to a `TCall` structure (page 3-72) that specifies the address of the peer that has accepted the connection, the value of options proposed using the `sndCall` parameter, and any data transmitted by the peer accepting the connection.

This parameter is only meaningful for synchronous calls to the `OTConnect` function.

DESCRIPTION

If the endpoint is in synchronous mode, the `OTConnect` function returns after the connection is established and fills in the fields of the `TCall` structure (referenced by the `rcvCall` parameter) with the actual values associated with this connection. These might be different from the values you specified using the `sndCall` parameter.

If the `OTConnect` function returns with the `kOTLookErr` result, this might be either because of a pending `T_LISTEN` or `T_DISCONNECT` event. That is, either a connection request from another endpoint has interrupted execution of the function, or the remote endpoint has rejected the connection. If you don't have a notifier installed, you can call the `OTLook` function to identify the event that caused the `kOTLookErr` result. If the event is `T_LISTEN`, you must accept or reject the incoming request and then continue processing the `OTConnect` function by calling `OTRcvConnect`. If the event is `T_DISCONNECT`, you must call the `OTRcvDisconnect` function to clear the error condition—that is, to deallocate memory and place the endpoint in the correct state.

If the endpoint is in asynchronous mode, the `OTConnect` function returns before the connection is established with a `kOTNoDataErr` result to indicate that the connection is in progress. When the connection is established, the endpoint provider calls your notifier, passing `T_CONNECT` for the `code` parameter. In response, you must call the `OTRcvConnect` function to read the connection parameters that would have been returned using the `rcvCall` parameter if the endpoint were in synchronous mode.

It is possible that the remote address returned in the `addr` field of the `rcvCall` parameter is not the same as the address you requested using the `sndCall->addr` field. This happens when the connection is accepted for a different endpoint than the one receiving the connection request.

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

SPECIAL CONSIDERATIONS

Not all endpoints support the sending of data with a connection request. Examine the `connect` field of the `TEndpointInfo` structure for the endpoint to

determine if the endpoint supports the sending of data and to determine the maximum size of the data.

VALID STATES

`T_IDLE`

SEE ALSO

You can use the `OTLook` function (page 3-95) to retrieve a pending asynchronous event.

You use a `TCall` structure (page 3-72) to specify the address of the remote peer, any data transmitted when establishing a connection, and any options for the connection.

You use the `OTRcvDisconnect` function (page 3-161) to acknowledge that your request for a connection has been rejected.

For information on how to use this function with a TCP/IP protocol, see page 8-17 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-11 in the ADSP chapter and page 15-10 in the PAP chapter.

You examine the `connect` field of the `TEndpointInfo` structure (page 3-58) to determine whether your endpoint supports the sending of data with a connection request.

OTRcvConnect

Reads the status of an outstanding or completed asynchronous call to the `OTConnect` function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint initiating the connection.
`call`	A pointer to a `TCall` structure (page 3-72) that contains information about the newly established connection. When the `OTRcvConnect` function returns, it fills in this structure. You can set this parameter to `nil`, in which case no information is returned to you.

DESCRIPTION

You call the `OTRcvConnect` function to determine the status of a previously issued `OTConnect` call. If you want to retrieve information about the connection, you must allocate buffers for the `addr` field and, if required, the `opt` and `udata` fields before you make the call.

If the endpoint is synchronous and blocking, the `OTRcvConnect` function waits for the connection to be accepted or rejected. If the connection is accepted, the function returns with a `kOTNoError` result. If the connection is rejected, the function returns with a `kOTLookErr` result. In this case, you should call the `OTLook` function to verify that a `T_DISCONNECT` event is the reason for the `kOTLookErr`, and then you should call the `OTRcvDisconnect` function to clear the event.

If the endpoint is asynchronous or nonblocking, the `OTRcvConnect` function returns with the `kOTNoDataErr` result if the connection has not yet been established.

VALID STATES

`T_OUTCON`

SEE ALSO

You use the `OTConnect` function (page 3-131) to request a connection request.

You use the `TCall` structure (page 3-72) to store information about the newly established connection.

You use the `OTLook` function (page 3-95) to retrieve pending asynchronous events.

You use the `OTRcvDisconnect` function (page 3-161) to acknowledge that your request for a connection has been rejected.

For information on how to use this function with a TCP/IP protocol, see page 8-17 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-11 in the ADSP chapter and page 15-10 in the PAP chapter.

You examine the `connect` field of the `TEndpointInfo` structure (page 3-58) to determine whether your endpoint supports the sending of data with a connection request.

OTListen

Listens for an incoming connection request.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint listening for the connection request.

`call`	A pointer to a `TCall` structure (page 3-72) that contains information about the address of the peer requesting the connection, option information, data associated with the connection request, and the connection ID for this connection.

DESCRIPTION

You use the `OTListen` function to listen for incoming connection requests. On return, the function fills in the `TCall` structure referenced by the `call` parameter with information about the connection request. After retrieving the connection request using the `OTListen` function, you can reject the request using the `OTSndDisconnect` function, or you can accept the request using the `OTAccept` function.

If the endpoint is in synchronous mode and is blocking, the `OTListen` function returns when a connection request has arrived. If the endpoint is in asynchronous mode or is not blocking, the `OTListen` function returns any pending connection requests or returns the `kOTNoDataErr` result if there are no pending connection requests. You can also call the `OTListen` function from within a notifier function in response to the `T_LISTEN` event. In this case, the function returns a result immediately.

SPECIAL CONSIDERATIONS

Not all endpoints support the sending of data with a connection request. Examine the `connect` field of the `TendpointInfo` structure for the endpoint to determine if the endpoint supports the sending of data and to determine the maximum size of the data.

To ensure portability, do not explicitly bind the endpoint to which you are passing off a connection if its address is to be the same as that of the endpoint listening for connection requests.

VALID STATES

`T_IDLE`, `T_INCON`

SEE ALSO

You use the `OTAccept` function (described next) to read an incoming connection request that you have retrieved using the `OTListen` function.

You use the `TCall` structure (page 3-72) to store information about the address of the peer requesting the connection, option information, data associated with the connection request, and the connection ID for this connection.

You use the `OTSndDisconnect` function (page 3-159) to reject a connection request.

You specify the maximum number of outstanding connections for an endpoint when you bind the endpoint using the `OTBind` function (page 3-87).

For information on how to use this function with a TCP/IP protocol, see page 8-18 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-11 in the ADSP chapter and page 15-10 in the PAP chapter.

You examine the `connect` field of the `TEndpointInfo` structure (page 3-58) to determine whether your endpoint supports the sending of data with a connection request.

OTAccept

Accepts an incoming connection request.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the listening endpoint.
`resRef`	The endpoint reference of the endpoint accepting the connection.

`call`	A pointer to a `TCall` structure (page 3-72) that contains information about the address of the peer requesting the connection, option information, data associated with the connection request, and the connection ID for this connection.

DESCRIPTION

You use the `OTAccept` function to accept a request that you retrieved using the `OTListen` function. You can accept a connection on either the same or on a different endpoint than the one listening for connection request.

- If you accept the connection on the same endpoint (the values of the `ref` and `resRef` parameters are the same), there must be no other outstanding connection requests on that endpoint. Otherwise, the call to `OTAccept` fails and returns the `kOTIndOutErr` result.
- If you accept the connection on a different endpoint (the values of the `ref` and `resRef` parameters are different), you are not required to bind the endpoint accepting the request first. If the endpoint is not bound, the provider binds it to the same address as that of the endpoint receiving the connection request. If you want to bind it explicitly to that address, you must set the `reqAddr->qlen` field to 0 and the endpoint must be in the `T_IDLE` state before calling the `OTAccept` function. If you want to bind it to a different address, there are no restrictions on the value of the `eqAddr->qlen` field.

If you do not wish to accept the request, you must call the `OTSndDisconnect` function.

If the endpoint is in asynchronous mode, the `OTAccept` function returns immediately with a `kOTNoError` result, indicating that processing has begun and that the client will be notified when it is complete.

When processing is finished and the connection is opened, the provider for the endpoint specified by the `ref` parameter, calls that endpoint's notifier, passing `T_ACCEPTCOMPLETE` for the `code` parameter and `kInvalidEndpointRef` for the `cookie` parameter. The provider for the endpoint specified by the `resRef` parameter, calls that endpoint's notifier, passing `T_PASSCON` for the `code` parameter and `ref` for the `cookie` parameter. If you have accepted the connection on the same endpoint (`ref` and `resRef` are the same), the provider issues the `T_ACCEPTCOMPLETE` event first, and then the `T_PASSCON` event.

If you have not installed a notifier, you can poll the endpoint accepting the connection for a change of state to T_DATAXFER; the change of state happens when the connection is opened.

SPECIAL CONSIDERATIONS

In asynchronous mode, it is possible for the endpoint to issue the T_ACCEPTCOMPLETE event before the OTAccept function returns the kOTNoError result.

Not all endpoints support the sending of data with a connection request. Examine the connect field of the TEndpointInfo structure for the endpoint to determine if the endpoint supports the sending of data and the maximum size of the data.

The OTAccept function fails with the kOTLookErr error if there are indications (T_DISCONNECT or T_LISTEN) waiting to be received. This is because

Calling the OTAccept function 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 request cannot be acted on until the OTListen function has been called 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 as many outstanding connection requests as you indicate in the qlen field, issuing an accept request, and getting a T_LISTENCOMPLETE event before the T_ACCEPTCOMPLETE event returns to you.

COMPLETION EVENT CODES

T_ACCEPTCOMPLETE	0x20000003	The OTAccept function has completed. The cookie parameter of the notifier function contains the endpoint reference of the endpoint to which the connection has been passed.

VALID STATES

Endpoint specified by the ref parameter: T_INCON

Endpoint specified by the resRef parameter: T_IDLE or T_UNBND

SEE ALSO

You use the `OTListen` function (page 3-135) to read a connection request before calling the `OTAccept` function to accept the request.

You use the `TCall` structure (page 3-72) to store information about the address of the peer requesting the connection, option information, data associated with the connection request, and the connection ID for this connection.

You use the `OTBind` function (page 3-87) to bind the endpoint accepting the request explicitly and to specify the number of connection requests that can be outstanding for the endpoint.

You use the `OTSndDisconnect` function (page 3-159) to reject an incoming connection request.

For information on how to use this function with a TCP/IP protocol, see page 8-18 in the TCP/IP chapter.

You examine the `connect` field of the `TEndpointInfo` structure (page 3-58) to determine whether your endpoint supports the sending of data with a connection request.

Functions for Connection-Oriented Transactionless Endpoints

To use connection-oriented transactionless endpoints, you must first establish a connection, as described in the previous section, and then use the `OTSnd` and `OTRcv` functions described in this section to transfer data.

The `OTSnd` and `OTRcv` functions do not use a special data structure to describe the data being transferred. Rather, the `buf` parameter is used to point to the buffer holding the data and the `nbytes` parameter is used to specify the size of the data being sent. Because the endpoints are already connected, it is not necessary to specify a destination address. Equally, options are defined when the connection is established; therefore, it is not necessary to specify options when sending data.

OTSnd

Sends data to a remote peer.

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);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint sending data.
`buf`	A pointer to the data being sent. If you are sending data that is not stored contiguously, this is a pointer to an `OTData` structure that describes the first data fragment.
`nbytes`	A long specifying the number of bytes being sent. If you are sending data that is not stored contiguously, you must set this field to the `kNetbufDataIsOTData` constant.
`flags`	A long bitmapped variable specifying whether the data to be sent is expedited (`T_EXPEDITED`) and whether more data remains to be sent (`T_MORE`). To set both fields, use the `bitAND` operator.

DESCRIPTION

You use the `OTSnd` function to send data to a remote peer. Before you use this function, you must establish a connection with the peer.

If the `OTSnd` function succeeds, it returns an integer (`OSStatus`) specifying the number of bytes that were actually sent. If it fails, it returns a negative integer corresponding to a result code that indicates the reason for the failure.

You specify the data to be sent by passing a pointer to the data (`buf`) and by specifying the size of the data (`nbytes`). The maximum size of the data you can send is specified by the `tsdu` field of the `TEndpointInfo` structure for the endpoint.

Some protocols use expedited data for control or attention messages. To determine whether the endpoint supports this service, examine the `etsdu` field of the `TEndpointInfo` structure. A positive integer for the `etsdu` field indicates

the maximum size in bytes of expedited data that you can send. To send expedited data, you must set the `T_EXPEDITED` bit of the `flags` parameter.

If you want to break up the data sent into smaller logical units, you can set the `T_MORE` bit of the `flags` parameter to indicate that you are using additional calls to the `OTSnd` function to send more data that belongs to the same logical unit. To indicate that the last data unit is being sent, you must specify 0 for `nbytes` and turn off the `T_MORE` flag. This is the only circumstance under which it is permitted to send a zero-length data unit. If the endpoint does not support the sending of zero-length data, the `OTSnd` function fails with the `kOTBadDataErr` result.

If the endpoint is in blocking mode, the `OTSnd` function returns after it actually sends the data. If flow-control restrictions prevent its sending the data, it retries the operation until it is able to send it. If the endpoint is in nonblocking mode, the `OTSnd` function returns with the `kOTFlowErr` result if flow-control restrictions prevent the data from being sent. When the endpoint provider is able to send the data, it returns a `T_GODATA` event to let you know that it is possible to send data.

If the endpoint is in non-blocking or asynchronous mode, it is possible that only part of the data is actually accepted by the transport provider. In this case, the `OTSnd` function returns 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 is not returned if the endpoint is in blocking mode.

If an asynchronous event, such as a disconnect, occurs and interrupts the `OTSnd` function, `OTSnd` returns with the `kOTLookErr` result.

The following table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSnd` function.

	Blocking	**Nonblocking**
Synchronous	The function returns when the provider lifts flow-control restrictions.	The function returns immediately.

	Blocking	Nonblocking
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately.	The function returns immediately.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

SPECIAL CONSIDERATIONS

The `XTI_SNDLOWAT` option allows endpoints that support it to negotiate the minimum number of bytes that must have accumulated in the endpoint's internal send buffer before they are sent. If the endpoint you are using supports this option, you can negotiate a value using the `OTOptionManagement` function. Because you use the `OTOptionManagement` function to set this option, it affects all subsequent sends.

VALID STATES

`T_DATAXFER`, `T_INREL`

SEE ALSO

For information about transferring data, see "Using Connection-Oriented Transactionless Service," beginning on page 3-44.

You can examine the `TEndpointInfo` structure (page 3-58) to find out what kind of data you can send and its maximum size.

You use the `OTData` structure (page 3-62) to transfer noncontiguous data.

For information on how to use this function with a TCP/IP protocol, see page 8-18 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-11 in the ADSP chapter and page 15-11 in the PAP chapter.

For additional information about the `OTOptionManagement` function, see the chapter "Option Management" in this book.

OTRcv

Reads data sent using a connection-oriented transactionless protocol.

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);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint receiving data.
`buf`	A pointer to a memory location where the incoming data is to be copied. You must allocate this buffer before you call the function. If you are doing a no-copy receive, this field is a pointer to an `OTBuffer` pointer.
`nbytes`	A long specifying the size of the buffer in bytes. If you are doing a no-copy receive, you must set this field to the `kNetbufDataIsOTBufferStar` constant.
`flags`	A long bitmapped variable specifying, on return, whether the data being sent is expedited (`T_EXPEDITED`) and whether more data remains to be received (`T_MORE`).

DESCRIPTION

You call the `OTRcv` function to read data sent by the peer to which you are connected. If the `OTRcv` function succeeds, it returns an integer (`OTStatus`) specifying the number of bytes received. The function places the data read into the buffer referenced by the `buf` parameter. If the function fails, it returns a negative integer corresponding to a result code that indicates the reason for the failure. You can call this function to receive either normal or expedited data. If the data is expedited, the `T_EXPEDITED` flag is set in the `flags` parameter.

If T_MORE is set in the flags parameter when the function returns, this means that the buffer you allocated is too small to contain the data to be read and that you must call the OTRcv function again. If you have read x bytes with the first call, the next call to the OTRcv function begins to read at the (x + 1) byte. Of course, if you need it, you must copy the data in the buffer to another location before calling the function again. Each call to this function that returns with the T_MORE flag set means that you must call the function again to get more data. When you have read all the data, the OTRcv function returns 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. To determine whether the endpoint supports TSDUs, examine the tsdu field of the TEndpointInfo structure. A value of T_INVALID means that the endpoint does not support it.

Some protocols allow you to send zero-length data to signal the end of a logical unit. In this case, if you request more than 0 bytes when calling the OTRcv function, the function returns 0 bytes only to signal the end of a TSDU.

If the OTRcv function returns and the T_EXPEDITED bit is set in the flags parameter, this means that you are about to read expedited data. If the number of bytes of expedited data exceeds the number of bytes you specified in the reqCount parameter, both the T_EXPEDITED and the T_MORE bits are set. You must call the OTRcv function until the T_MORE flag is not set to retrieve the rest of the expedited data.

If you are calling the OTRcv function repeatedly to read normal data and a call to the function returns T_EXPEDITED in the flags parameter, the next call to the OTRcv function that returns without the T_EXPEDITED flag set returns normal data at the place it was interrupted. It is your responsibility to remember where that was and to continue processing normal data. You can determine how much normal data you read by maintaining a running total of the number of bytes returned in the OTStatus result.

If the endpoint is in asynchronous mode or is not blocking, the function returns with the kOTNoDataErr result if no data is available. If you have installed a notifier, the endpoint provider calls your notifier and passes T_DATA or T_EXDATA for the code parameter when there is data available. If you have not installed a notifier, you may poll for these events using the OTLook function. Once you receive a T_DATA or T_EXDATA event, you should continue in a loop, calling the OTRcv function until it returns with the kOTNoDataErr result.

If the endpoint is in synchronous mode and is blocking, the endpoint waits for data if none is currently available. You should avoid calling the OTRcv function this way because it might cause processing to hang if no data is available. If

you are doing other operations in synchronous mode, you should put the endpoint in nonblocking mode before calling the `OTRcv` function.

SPECIAL CONSIDERATIONS

You should be prepared for a `T_DATA` event and then a `kOTNoDataErr` error when you call the `OTRcv` function. This seems unusual, but it can occur if you are calling `OTRcv` in the foreground when a `T_DATA` event comes in.

Whenever the `OTRcv` function returns a `kOTLookErr` error, it is very important that you call the `OTLook` function. 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 you will hang waiting. 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 `XTI_RCVLOWAT` option allows endpoints that support it to negotiate the minimum number of bytes that must have accumulated in the endpoint's internal receive buffer before the endpoint provider generates a `T_DATA` event. If the endpoint you are using supports this option, you can negotiate a value using the `OTOptionManagement` function. Because you use the `OTOptionManagement` function to set this option, it affects all subsequent sends.

VALID STATES

`T_DATAXFER`, `T_OUTREL`

SEE ALSO

You use the `OTLook` function (page 3-95) to poll for the `T_DATA` or `T_EXDATA` events.

You use the `OTIsNonBlocking` function, described in the reference section of the chapter "Providers" in this book, to determine the current operational mode of the endpoint. It is recommended that the endpoint be in nonblocking mode before you call the `OTRcv` function.

For information on how to use this function with a TCP/IP protocol, see page 8-19 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-12 in the ADSP chapter and page 15-11 in the PAP chapter.

You use the `OTOptionManagement` function, described in the chapter "Option Management" in this book, to negotiate the `XTI_RCVLOWAT` option.

Functions for Connection-Oriented Transaction-Based Endpoints

After you establish a connection, you can transfer data using connection-oriented transaction-based endpoints by calling the `OTSndRequest` function to send a request, the `OTRcvRequest` function to read a request, the `OTSndReply` function to reply to the request, and the `OTRcvReply` function to read the reply. This section also describes the `OTCancelRequest` function, which you use to cancel an outgoing request, and the `OTCancelReply` function, which you use to cancel an incoming request.

Connection-oriented transaction-based service used by protocols such as ADSP is described at greater length in the section "Using Connection-Oriented Transaction-Based Service," beginning on page 3-50. This section describes the functions used to send and retrieve requests and replies.

OTSndRequest

Sends a request to a connection-oriented transaction-based responder.

C INTERFACE

```
OSStatus OTSndRequest(EndpointRef ref, TRequest* req,
                      OTFlags reqFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndRequest(TRequest* req, OTFlags reqFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint making the request.

req	A pointer to a TRequest structure (page 3-76) that contains information about the request, options for this request, and the transaction ID of the request.
reqFlags	A bitmapped long specifying whether you are sending request data using additional calls to this function (T_MORE) and whether you plan to acknowledge replies (T_ACKNOWLEGED). Use the bitAND operator to set both bits.

DESCRIPTION

You use the OTSndRequest function to initiate a transaction. When the responding peer replies to your request, you use the OTRcvReply function to read the reply.

By default, delivery is guaranteed for you, but not for the responder. That is, you will always find out whether your request was received, but the responder only receives acknowledgment that you received the reply if you set the T_ACKNOWLEDGED bit in the reqFlags parameter when you send the request.

If the responder is an Open Transport endpoint, its provider generates a T_REPLYCOMPLETE event when you have read the reply. This happens whether or not the T_ACKNOWLEDGED bit is set; but if it is set, this guarantees that the reply was delivered. If you don't set this flag, the responder's call to the OTSndReply function returns right away, and the responding endpoint receives no additional information as to whether the reply was received and the data was read.

Setting the T_MORE bit tells the endpoint provider that you are using several calls to the OTSndRequest function to send the request data. Note that even though you are using several calls, the request data, put all together, must still not exceed the value specified for the etsdu field in the endpoint's TEndpointInfo structure.

If the endpoint is in blocking mode and flow-control restrictions prevent the endpoint provider from accepting the OTSndRequest function, Open Transport retries the operation until flow-control restrictions are lifted.

If the endpoint is in nonblocking mode and flow-control restrictions prevent the endpoint provider from accepting the OTSndRequest function, Open Transport returns the kOTFlowErr result. When flow-control restrictions are lifted, the provider issues a T_GODATA event, which you can retrieve using your notifier function or by polling the endpoint using the OTLook function. When you get this event, you can try sending the request again.

The next table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSndRequest` function.

	Blocking	Nonblocking
Synchronous	The function returns when the provider lifts flow-control restrictions and the request has been sent to the protocol.	The function returns if flow-control restrictions are in effect or the request data has been sent to the protocol.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately.	The function returns immediately.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

VALID STATES

`T_DATAXFER`, `T_INREL`

SEE ALSO

You use the `OTRcvReply` function (page 3-154) to read the reply to your request.

You use the `TRequest` structure (page 3-76) to specify information about the request, options for this request, and the transaction ID of the request.

The maximum size of a request is defined by the `etsdu` field of the `TEndpointInfo` structure (page 3-58).

You use the `OTLook` function (page 3-95) to retrieve pending asynchronous events.

You use the `OTData` structure (page 3-62) to describe noncontiguous data.

OTRcvRequest

You use the `OTRcvRequest` function to read a request from a connection-oriented transaction-based requester.

C INTERFACE

```
OSStatus OTRcvRequest(EndpointRef ref, TRequest* req,
                      OTFlags* reqFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvRequest(TRequest* req, OTFlags* flags);
```

PPARAMETERS

`ref`	The endpoint reference of the endpoint reading the request.
`req`	A pointer to a `TRequest` structure (page 3-76) that contains information, on return, about the incoming request. See the description of the `TRequest` structure for how to set this parameter when doing a no-copy receive.
`reqFlags`	A bitmapped long specifying, on return, whether there is more request data coming (`T_MORE`) and whether the provider is going to acknowledge replies (`T_ACKNOWLEDGED`).

DESCRIPTION

You use the `OTRcvRequest` function to read an incoming request. After reading the request, you can use the `OTSndReply` function to reply to that request or the `OTCancelRequest` function to reject the request.

When the `OTRcvRequest` function returns, the `req->data.buf` field points to the request data and the `req->sequence` field specifies a transaction ID for this transaction. You must use this same sequence value when calling the `OTSndReply` function to reply to this request or the `OTCancelRequest` function to reject it.

If you have allocated a buffer that is too small to hold the request data, the provider sets the `T_MORE` bit in the `reqFlags` field to indicate that there is more request data to be read. You must call the `OTRcvRequest` function until the `T_MORE` flag is cleared in order to retrieve the rest of the request data. The `req->opt` field contains returns no information for these additional calls.

If the endpoint is in synchronous mode and is blocking, the `OTRcvRequest` function returns only when a request arrives. If the endpoint is asynchronous

or is not blocking, the OTRcvRequest function returns either the next unread request or the kOTNoDataErr result if there are no pending requests.

If flow-control restrictions prevent the provider from accepting the data when you call the OTRcvRequest function, the function returns the kOTFlowErr result. The provider issues the T_GODATA event when flow-control restrictions are lifted.

When a request arrives, the provider generates a T_REQUEST event. You can poll for this event using the OTLook function or call the function for as long as the kOTNoDataErr result is returned. If you have a notifier installed for this endpoint, the event is sent to the notifier.

VALID STATES

T_DATAXFER, T_OUTREL

SEE ALSO

You use the OTSndReply function (described next) to reply to a request that you have read using the OTRcvRequest function.

You can use the OTLook function to poll for T_REQUEST events (page 3-95).

OTSndReply

You use the OTSndReply function to reply to a connection-oriented transaction-based request.

C INTERFACE

```
OSStatus OTSndReply(EndpointRef ref, TReply* reply,
                    OTFlags* replyFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndReply(TReply* reply, OTFlags flags);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint reading the request.
`reply`	A pointer to a `TReply` structure (page 3-77) that specifies the reply data being sent, the transaction ID for this transaction, and any options you want to set.
`replyFlags`	A bitmapped long specifying whether the rest of the reply is being sent with a subsequent call to this function (`T_MORE`) or whether this is the complete reply (`T_MORE` not set).

DESCRIPTION

You use the `OTSndReply` function to reply to a request you have read using the `OTRcvRequest` function. The `reply` parameter contains the reply to be sent, and the `replyFlags` parameter specifies whether you are sending the entire reply with this send (`T_MORE` bit clear) or sending just part of the reply (`T_MORE` bit set). If you are using multiple sends to send the reply, you must set the `T_MORE` bit on each but the last send. The total size of the data you send using multiple sends must not exceed the value of the `tsdu` field of the `TEndpointInfo` structure for this endpoint.

If the endpoint is in blocking mode, the `OTSndReply` function returns after it has sent the reply. If the endpoint is in nonblocking mode, the `OTSndReply` function returns the `kOTFlowErr` result if the endpoint provider is unable to send the reply because of flow-control restrictions. The provider issues the `T_GODATA` event when these restrictions are lifted. You can use the `OTLook` function to poll for this event, or you can use your notifier to handle it.

If the endpoint is in asynchronous mode, the provider calls your notifier when the `OTSndReply` function completes. The `code` parameter of the notifier function contains the `T_REPLYCOMPLETE` event, the `cookie` parameter contains the `reply` parameter passed with the `OTSndReply` function, and the `result` parameter contains the function result.

The next table shows how the endpoint's mode of execution and blocking status affects the behavior of the `OTSndReply` function.

	Blocking	Nonblocking
Synchronous	The function returns when the provider lifts flow-control restrictions and the reply has been successfully sent or timed out.	The function returns if flow-control restrictions are in effect or when the reply has been successfully sent or timed out.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.
Asynchronous	The function returns immediately.	The function returns immediately.
	The provider calls your notifier, passing `T_REPLYCOMPLETE` for the `code` parameter when the reply is successfully sent or timed out.	The provider calls your notifier, passing `T_REPLYCOMPLETE` for the `code` parameter when the reply is successfully sent or timed out.
	The `kOTFlowErr` result is never returned.	The `kOTFlowErr` result might be returned.

VALID STATES

`T_DATAXFER`, `T_OUTREL`

COMPLETION EVENTS

`T_REPLYCOMPLETE`	0x20000004	The `OTSndReply` function has completed. The `cookie` parameter of the notifier function points to the `reply` parameter.

SEE ALSO

You use the `OTRcvRequest` function (page 3-149) to read an incoming request before calling the `OTSndReply` function to reply to the request.

You use the `TReply` structure (page 3-77) to specify the reply data being sent, the transaction ID for this transaction, and any options you want to set.

The peer endpoint calls the `OTRcvReply` function (page 3-154) to acknowledge receiving the reply you send using the `OTSndReply` function.

You use the `OTData` structure (page 3-62) to transfer noncontiguous data.

OTRcvReply

Reads a transaction reply sent by a connection-oriented responder.

C INTERFACE

```
OSStatus OTRcvReply(EndpointRef ref, TReply* reply,
                    OTFlags* replyFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvReply(TReply* reply, OTFlags* flags);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint reading the reply.
`reply`	A pointer to a `TReply` structure (page 3-77) that specifies the size and location of buffers into which the function, on return, stores data, option information, and the ID of the transaction. See the description of the `TReply` structure for how to set this parameter when doing a no-copy receive.
`replyFlags`	A long bitmapped field specifying `T_MORE` or `T_PARTIALDATA`. A value of `T_MORE` indicates that the buffer pointed to by `reply->data.buf` is too small to contain the reply. A value of `T_PARTIALDATA` indicates that the data unit being read does not contain the complete reply and that the next data unit might belong to a different transaction.

DESCRIPTION

You use the `OTRcvReply` function to read the reply to a request that you sent using the `OTSndRequest` function.

If the endpoint is in asynchronous mode, the endpoint provider issues the `T_REPLY` event to let you know that incoming reply data is available. After you retrieve this event (using the `OTLook` function or your notifier function,) you must call the `OTRcvReply` function repeatedly to read the reply data until it returns `kOTNoDataErr`. The endpoint provider does not generate additional `T_REPLY` events until you have read the complete reply.

If a transaction has timed out awaiting reply data, the `OTRcvReply` function returns a `kETIMEDOUTErr` result; the `sequence` field of the `reply` parameter specifies which request has timed out.

If you have issued multiple requests, it is not possible to know ahead of time how incoming replies match your requests. You must be prepared to receive a reply to any outstanding request. One way to manage this situation is to call the `OTRcvReply` function with the `reply->udata.maxlen` field set to 0. The rest of the information returned by the function on this first call lets you know the sequence number of the reply as well as the `replyFlags` setting. Once you determine the matching request and the appropriate reply buffer, you can call the `OTRcvReply` function a second time to read the actual reply data. On the second and subsequent reads, Open Transport sets the `reply->opt.len` field to 0. It is guaranteed that once a reply has been partially read, subsequent calls to `OTRcvReply` read from that same reply until all the data has been read.

If the `T_MORE` bit is set in the `flags` parameter, this means your buffer is not large enough to hold the entire reply. You must call the `OTRcvRequest` function again to retrieve more request data. Open Transport ignores the `addr` and `opt` fields of the `reply` parameter for subsequent calls to the function. The `T_MORE` flag is not set for the last reply packet to let you know that this is the last packet.

If the `T_PARTIALDATA` bit is set in the `flags` parameter, this means that the data you are about to read with the `OTRcvReply` function does not constitute the entire reply and that you must call the function again to read more of or the rest of the reply.

If the `T_MORE` and the `T_PARTIALDATA` bits are both set, this means that the data you are about to read constitutes only part of the reply and that your buffer is too small to contain even this chunk. In this case, you must call the function again until the `T_MORE` flag is clear. The `T_PARTIALDATA` bit is set only on the first call to the function.

If you are communicating with multiple responders and if the `OTRcvUReply` function returns with the `T_PARTIALDATA` flag set, it is possible that your next call to the function might not read the rest of the reply because the next data unit coming in belongs to a different reply. One way to handle this situation is to use the next call to the `OTRcvReply` function to determine the sequence number of the incoming reply (by setting `req->udata.maxlen` to 0) and then, having determined which reply data is coming in, read the data into the appropriate buffer.

VALID STATES

`T_IDLE`

SEE ALSO

The request to which you are receiving a reply is defined by a previous call to the `OTSndRequest` function (page 3-147).

You use the `TReply` structure (page 3-77) to specify the size and location of buffers into which the function, on return, stores data, option information, and the ID of the transaction.

You use the `OTLook` function (page 3-95) to poll for asynchronous events.

OTCancelRequest

Cancels an outstanding request as defined by a call to the `OTSndRequest` function.

C INTERFACE

```
OSStatus OTCancelRequest(EndpointRef ref, OTSequence seq);
```

C++ INTERFACE

```
OSStatus Tendpoint::CancelRequest(OTSequence seq);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint that has sent the request being cancelled.
`seq`	A long, specifying the transaction ID of the request being canceled. You must specify the same value that you used for the `sequence` field of the `req` parameter you passed to the `OTSndRequest` function. If you specify 0 for this parameter, the provider cancels all outstanding requests. If you specify an invalid sequence number, the provider does not do anything.

DESCRIPTION

When you make a call to the `OTSndRequest` function, the endpoint provider allocates memory for internal buffers for this transaction. If you are no longer interested in the transaction, you must tell the endpoint provider by calling the `OTCancelRequest` function. Explicitly canceling a request allows the provider to free up the memory associated with a transaction request.

If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment. It is your responsibility to deallocate memory that you have reserved for the address, options, and data buffers associated with the canceled function.

Use `OTCancelRequest` to cancel an *outgoing* request; use `OTCancelReply` to cancel an *incoming* request.

VALID STATES

`T_IDLE`

SEE ALSO

You use the `OTSndRequest` function (page 3-147) to send a request.

You use the `OTCancelReply` function (described next) to cancel an incoming request.

OTCancelReply

Cancels an outstanding call to the `OTRcvRequest` function.

C INTERFACE

```
OSStatus OTCancelReply(EndpointRef ref, OTSequence seq);
```

C++ INTERFACE

```
OSStatus TEndpoint::CancelReply(OTSequence seq);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint that has sent the request being canceled.
`seq`	A long, specifying the transaction ID of the request being canceled. You must specify the same value that was passed to you in the `seq` field of the `req` parameter to the `OTRcvRequest` function. If you specify 0 for this parameter, the provider cancels all outstanding incoming requests. If you specify an invalid sequence number, the provider does not do anything.

DESCRIPTION

When you make a call to the `OTRcvRequest` function, the provider allocates memory for internal buffers and assigns a sequence value to identify this transaction. If you are no longer interested in a transaction, you must explicitly cancel the transaction by calling the `OTCancelReply` function. Calling this function allows the provider to free up the memory it has reserved and to reuse the sequence number associated with the canceled transaction.

If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment. It is your responsibility to deallocate memory that you have reserved for the address, options, and data buffers associated with the cancelled `OTRcvRequest` function.

Use the `OTCancelReply` function to cancel an *incoming* request; use the `OTCancelRequest` function to cancel an *outgoing* request.

VALID STATES

`T_IDLE`

SEE ALSO

You use the `OTSndRequest` function (page 3-147) to send a request.

You use the `OTCancelRequest` function (page 3-156) to cancel an outgoing request.

Tearing Down a Connection

You use the functions described in this section to tear down a connection. Depending on the circumstances, you might use the `OTSndDisconnect` function to initiate an abortive disconnect or the `OTSndOrderlyDisconnect` function to initiate an orderly disconnect. If you are responding to a disconnection request, you call the `OTRcvDisconnect` function to acknowledge an abortive disconnect or the `OTRcvOrderlyDisconnect` function to acknowledge an orderly disconnect. You can also use the `OTSndDisconnect` function to reject an incoming connection request.

OTSndDisconnect

Tears down an open connection (abortive disconnect) or rejects an incoming connection request.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference for the endpoint tearing down the connection or rejecting the connection request.
`call`	A pointer to a `TCall` structure (page 3-72) that specifies the connection to be torn down or rejected and specifies data sent with the disconnection request if the endpoint supports sending such data.

DESCRIPTION

There are two functions that you can use to tear down a connection: `OTSndDisconnect` for an abortive disconnect, or `OTSndOrderlyDisconnect` for an orderly disconnect. It is recommended that you use the `OTSndOrderlyDisconnect` function for tearing down a connection whenever possible and that you use the `OTSndDisconnect` function only for rejecting incoming connection requests.

If the endpoint is in asynchronous mode, the `OTSndDisconnect` function returns immediately with a result of `kOTNoError` to indicate that the disconnection process has begun and that your notifier function will be called when the process completes.

When the connection has been broken, the provider issues a `T_DISCONNECTCOMPLETE` event. If you have installed a notifier function, Open Transport calls your notifier and passes this event in the `code` parameter. The `cookie` parameter contains the `call` parameter. If you have not installed a notifier function, you cannot determine when this function completes.

COMPLETION EVENT CODES

T_DISCONNECTCOMPLETE	0x20000005	The OTSndDisconnect function has completed. The cookie parameter contains 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)

SEE ALSO

To acknowledge an abortive disconnect, you call the OTRcvDisconnect function (described next).

You use the TCall structure (page 3-72) to describe the connection being torn down or rejected.

You use the OTListen function (page 3-135) to listen for a disconnection request.

You can examine the discon field of the TEndpointInfo structure (page 3-58) for the endpoint to determine whether the endpoint supports sending data during the disconnection and to find out the maximum size of such data.

For information on how to use this function with a TCP/IP protocol, see page 8-19 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-12 in the ADSP chapter and page 15-11 in the PAP chapter.

For information on abortive and orderly disconnects, see "Terminating a Connection," beginning on page 3-35.

OTRcvDisconnect

Identifies the cause of a connection break or of a connection rejection, acknowledges and clears the corresponding disconnection event.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint receiving the disconnection request.
`discon`	A pointer to a `TDiscon` structure that specifies any user data, a reason for the disconnection, and a connection request sequence number.

DESCRIPTION

Calling the `OTRcvDisconnect` function clears the corresponding disconnection event and retrieves any user data sent with the disconnection.

If you do not care about data returned with the disconnection and do not need to know the reason for the disconnection nor the sequence ID, you may specify a `nil` pointer for the `discon` parameter. In this case, the provider discards any user data associated with the disconnection.

The `OTRcvDisconnect` function behaves in the same way for all modes of operation. If there is no disconnection request pending, the function returns with the `kOTNoDisconnectErr` result. If there is a disconnection request pending, the function returns either the `kOTNoError` or `kOTBufferOverflowErr` result. In the latter instance, you need to check the `discon` field of the `TEndpointInfo` structure for your endpoint and make sure that the buffer referenced by the `udata.buf` field is at least as big as the value specified for the `discon` field.

VALID STATES

`T_DATAXFER`, `T_OUTCON`, `T_OUTREL`, `T_INREL`, `T_INCON` (when there is more than one pending disconnection request)

SEE ALSO

To send an abortive disconnect, you call the `OTSndDisconnect` function (page 3-159).

You use the `TDiscon` structure (page 3-79) to specify that provides user data, a reason for the disconnection, and a connection request sequence number.

For information on how to use this function with a TCP/IP protocol, see page 8-19 in the TCP/IP chapter.

For information on how to use this function with AppleTalk protocols, see page 13-12 in the ADSP chapter and page 15-11 in the PAP chapter.

For information on abortive and orderly disconnects, see "Terminating a Connection," beginning on page 3-35.

You examine the `discon` field of the `TEndpointInfo` structure (page 3-58) to find out whether your endpoint supports sending data with the disconnection request and to determine the maximum size of such data.

OTSndOrderlyDisconnect

Initiates or completes an orderly disconnection.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint initiating the orderly disconnect.

DESCRIPTION

You call the `OTSndOrderlyDisconnect` function to initiate an orderly release of a connection and to indicate to the peer endpoint that you have no more data to send. After calling this function, you must not send any more data over the connection. However, you can still continue to receive data if the peer endpoint has not yet called the `OTSndOrderlyDisconnect` function.

This function is a service that is not supported by all protocols. If it is supported, the `servtype` field of the `TEndpointInfo` structure has the value `T_COTS_ORD` or `T_TRANS_ORD`.

The `OTSndOrderlyDisconnect` function behaves exactly the same in all modes of operation.

VALID STATES

`T_DATAXFER`, `T_INREL`

SEE ALSO

To send an abortive disconnect or to reject a connection request, you call the `OTSndDisconnect` function (page 3-159).

For information on abortive and orderly disconnects, see "Terminating a Connection," beginning on page 3-35.

You examine the `TEndpointInfo` structure (page 3-58) to determine whether the endpoint supports orderly release.

OTRcvOrderlyDisconnect

Acknowledges a request for an orderly disconnect.

C INTERFACE

```
OSStatus OTRcvOrderlyDisconnect(EndpointREf ref);
```

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint acknowledging receipt of the disconnect request.

DESCRIPTION

The `OTRcvOrderlyDisconnect` function is a service that is not supported by all protocols. If it is, the `servtype` field of the `TEndpointInfo` structure has the value `T_COTS_ORD` or `T_TRANS_ORD` for the endpoint.

After using the `OTRcvOrderlyDisconnect` function to acknowledge receipt of a disconnection request, there will not be any more data to receive. Calls to the `OTRcv` function (for a transactionless connection) or to the `OTRcvRequest` function (for a transaction-based connection) after acknowledging a disconnection request fail with the `kOTOutStateErr` result. If the endpoint supports a remote orderly disconnect, you can still send data over the connection if you have not yet called the `OTSndOrderlyDisconnect` function.

The `OTRcvOrderlyDisconnect` function behaves in the same way in all modes of operation. If there is no disconnection request pending, the function returns with the `kOTNoReleaseErr` result. It there is a disconnection request pending, the function returns either the `kOTNoError` or `kOTBufferOverflowErr` result. In the latter instance, you need to check the `discon` field of the `TEndpointInfo` structure for your endpoint and make sure that the buffer referenced by the `udata.buf` field is at least as big as the value specified for the `discon` field.

VALID STATES

`T_DATAXFER`, `T_OUTREL`

SEE ALSO

You use the `OTSndOrderlyDisconnect` function (page 3-163) to send an orderly disconnect.

For information on abortive and orderly disconnects see "Terminating a Connection," beginning on page 3-35.

You examine the `TEndpointInfo` structure (page 3-58) to determine whether the endpoint supports orderly release.

Mappers

Contents

This chapter describes mappers, the type of Open Transport provider that lets your application map entity names to protocol addresses. You can use mapper functions to register a name, to look up a name or name pattern, or to remove a registered name. Which functions are supported depends on the name-registration protocol underlying the mapper provider you create. For more detailed information about how mapper functions are implemented for the protocol you are interested in, consult the documentation provided for that protocol.

You do not have to open a mapper provider if you are interested only in registering a name or looking up an address corresponding to a name.

- If the protocol you are using allows you to bind an endpoint by name and you do so, the name is automatically registered on the network. This is a more efficient way to register a name on the network than to create a mapper to do it.
- If you want to obtain the address that corresponds to an entity name, you can use the endpoint function `OTResolveAddress`. Using this function also saves you the trouble of opening a mapper. However, you cannot use this function to look up a name pattern; that is, the name you look up cannot include a wildcard character.

If you are using an endpoint that cannot be bound by name, if you want to look up a name pattern, or if you want to use other mapper functions, you need to read this chapter and learn how to create a mapper provider.

This chapter begins with a general description of mapper providers and continues with a more detailed discussion of how you use mappers asynchronously and how you use the mapper to look up names. The functions used to register names and delete names are discussed in the section "Mappers Reference," beginning on page 4-12.

Mapper providers, like all Open Transport providers, can operate synchronously or asynchronously, can block, and can acknowledge sends. For general information about Open Transport providers, see the chapter "Providers" earlier in this book.

About Mappers

A **mapper** is a communications path between your application and a **mapper provider,** which is a protocol that allows you to map a name to a network address, if the underlying protocol allows it, and to register that name-address pair so that it becomes visible to all other entities on a network. When you create a mapper, you instantiate a data structure that contains information about the mapper provider's mode of operation, information about the mapper's state, and pointers to mapper functions. These functions are your application's interface to the underlying name-registration protocol. Which functions you use depends on the name-registration protocol you select when you create a mapper. For example, if you select the AppleTalk Name-Binding Protocol (NBP), which supports dynamic name and address registration, you can use all the mapper functions described in this chapter: you can register a name, look up a name, and remove a registered name. If you select the TCP/IP protocol family, which uses the domain name resolver (DNR), this choice does not support dynamic name and address registration, and you can only look up a name that has been registered using other means.

When you create a mapper, you specify which protocol is to provide the name-registration service. You also have the option of specifying the layers of protocols underlying that service; these layers provide basic data-transfer services. For example, AppleTalk's NBP protocol relies on the more basic Datagram Delivery Protocol (DDP) to transfer data as required for name registration and name lookup. You do not have to specify these underlying data-transfer protocols. When you select the name-registration protocol you are interested in, a default configuration is provided. For more information about the default configuration for your protocol, please consult the documentation furnished for that protocol. Of course, as with any Open Transport provider, you do have the choice of specifying the underlying protocols, all the way down to the hardware link. For more information, see the chapter "Configuration Management" in this book.

When you create a mapper, you obtain a mapper reference. A **mapper reference,** like an endpoint reference, identifies the instance of the provider you have created. You must pass this reference as a parameter to all other mapper functions. You can open multiple mappers. For example, if you are writing a network administration application, you might want to create a mapper for each protocol used over the network. If you do open multiple mappers, the

mapper reference tells Open Transport which mapper is invoked for any one function call.

Like endpoint providers, mapper providers also have a state attribute, which helps Open Transport manage these providers. Unlike endpoints, however, mappers do not provide functions that allow you to determine their state. A mapper can be either in an uninitialized (`T_UNINIT`) state if it was closed by the system, or in the idle (`T_IDLE`) state after it has been opened.

Using Mappers

This section begins by describing how the general provider functions that govern a provider's mode of operation apply to mapper providers. It goes on to discuss information you need to know in order to use mapper functions: how you format names and addresses specified in parameters to mapper functions and how you handle processing when calling mapper functions asynchronously. This section concludes with a discussion of different techniques you can use when using the mapper to search for a name pattern.

Setting Modes of Operation for Mappers

Like all Open Transport providers, mappers can use different modes of operation. A mapper can execute synchronously or asynchronously. You set the mapper's default mode of execution by using the appropriate function to open it; for example, you can open a mapper that executes asynchronously by calling the `OTAsyncOpenMapper` function to create the mapper. After opening the mapper, you can change its mode of execution by calling the `OTSetSynchronous` or `OTSetAsynchronous` functions. To determine how mapper functions execute, you call the `OTIsSynchronous` function. A mapper uses one asynchronous event and four completion events. Table 4-1 lists the event codes that the mapper provider can pass to your application and explains the meaning of the `cookie` parameter to the notifier for each function. For more detailed information, see the descriptions of the mapper functions beginning on page 4-19.

Table 4-1 Completion events for asynchronous mapper functions

Completion code	Meaning
T_OPENCOMPLETE	The OTAsyncOpenMapper function has completed. The cookie parameter contains the mapper reference.
T_REGNAMECOMPLETE	The OTRegisterName function has completed. The cookie parameter contains the reply parameter, unless it was NULL, in which case it contains the request parameter.
T_DELNAMECOMPLETE	The OTDeleteName or the OTDeleteNameByID functions have completed. For the OTDeleteName function, the cookie parameter holds a pointer to the name parameter. For the OTDeleteNameByID function, the cookie parameter contains the id parameter.
T_LKUPNAMERESULT	The OTLookupName function has returned a name, but it has not yet completed because there are more names to retrieve.
T_LKUPNAMECOMPLETE	The OTLookupName function has completed. The cookie parameter contains the reply parameter.

The only way to cancel an asynchronous mapper function is to call the OTCloseProvider function, passing the mapper reference for which the function was executed. The OTCloseProvider function is described in the chapter "Providers" in this book.

By default, mappers do not block and do not acknowledge sends. You can change a mapper's blocking status by using the OTSetBlocking function. You can change a mapper's send-acknowledgment status by using the OTAckSends function. These functions are described in the chapter "Providers" in this book. Mapper providers are not affected by their send-acknowledgment status. However, a mapper provider's blocking status might affect the behavior of mapper functions. For example, if a mapper is blocking, heavy network traffic might cause mapper functions to wait before sending or receiving data. If a mapper is nonblocking and you are doing a lot of name lookups, the OTLookupName function might return with the kOTFlowErr result. In this case, you can try executing the function later.

Specifying Name and Address Information

Several mapper functions require that you specify a name or address. This might be a name to register or to look up. Specifying a name or address means that you have to create a buffer that contains the information and then create a `TNetbuf` structure that specifies the size and location of this buffer. The format that you use to store a name or an address is specific to the name-registration protocol that underlies the mapper and is exactly the same as the name and address formats that you can use to bind an endpoint. For information about name and address formats, please consult the documentation provided for the protocol you are using.

If the protocol supports it, you can specify a name pattern rather than a name when calling the `OTLookupName` function. Different protocols might use different wildcard characters to define name patterns. Please consult the documentation provided for your protocol to determine valid wildcard characters and how you use these to specify name patterns.

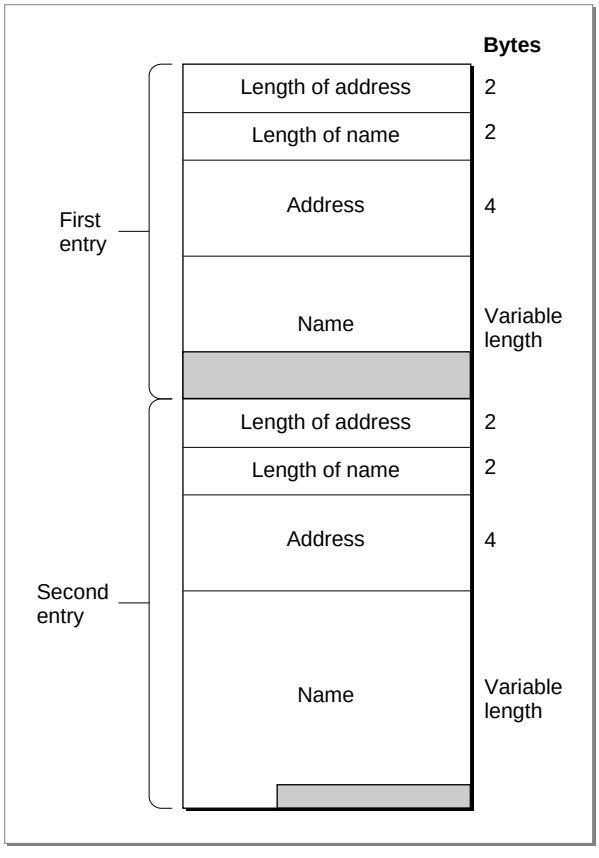
Searching for Names

You use the `OTLookupName` function to search for a registered name or for a list of names if your protocol supports name pattern matching. You use the `req` parameter to the function to specify the name or name pattern to search for. When the function returns, it uses the `reply` parameter to pass back the matching name or names.

The `req` parameter is a pointer to a `TLookupRequest` structure containing the name or name pattern to be found and additional information that the mapper can use in conducting the search. You use the `maxcnt` field to specify the number of names you expect to be returned. If you are looking for a specific name, set this field to 1. If you are looking for a name pattern, you can use this field to indicate the number of matches you expect the `OTLookupName` function to return. You use the `timeout` field to specify the amount of time (in milliseconds) available for this search. If a match is not found within the specified time, the function returns with the `kOTNoDataErr`. If you do not specify a value for the `maxcnt` field, or if the number you specify is larger than the number of names that match the given pattern, the mapper provider uses the value given in the `timeout` field to determine when to stop the search.

The `reply` parameter is a pointer to a `TLookupReply` structure that contains two fields. The `names` field describes the size and location of the buffer in which the replies are placed when the function returns; the `rspcount` field specifies the number of matching entries found. Figure 4-1 shows how the contents of a reply buffer containing two entries are stored.

Figure 4-1 Format of entries in `OTLookupName` reply buffer



The first 2 bytes of each entry specifies the length of the address; the second 2 bytes specifies the length of the name. The address is stored next and then the name, padded to a quad-word boundary. Given a pointer to the reply buffer

(replyBufPtr), you can obtain the length of the address (alen) and the length of the name (nlen), and then you can compute the length of an entry in the reply buffer as follows:

```
bufPtr = (short*)replyBufPtr
alen = ((UInt16*)bufPtr)[0];
nlen = ((UInt16*)bufPtr)[1];
len = alen + nlen + 4                    /* length of first entry */
```

Because the entry is aligned on a quad-word boundary, you must account for this padding to determine where the next entry begins. For example, the following formula computes the beginning of the next entry:

```
bufPtr = bufPtr + (len + 3L) & ~3L);
```

The next section "Retrieving Multiple Entries From the Reply Buffer," presents a small code sample that shows how to parse the reply buffer.

Retrieving Multiple Entries From the Reply Buffer

Listing 4-1 shows the sample routine DoParseOTLookup, which retrieves name-address entries from the reply buffer filled in by the OTLookupName function. The buffer being parsed in the sample listing contains name-address entries for AppleTalk endpoints.

Listing 4-1 Parsing the reply buffer for OTLookupName

```
void DoParseOTLookup(Ptr returnBufferPtr, long numFound)
{
    char nameString[100];
    DDPAddress ddpAddress;
    char* bp;
    long index;

    index = 0;
    bp = (char*) returnBufferPtr;
    while(index < numFound)
    {
        UInt16 len;             /* entry length */
        UInt16 alen;            /* adress length */
```

```
        UInt16 nlen;            /* name length */

        alen = ((UInt16*) bp)[0];
        nlen    = ((UInt16*) bp)[1];
        len     = alen + nlen + 4;

        BlockMove((Ptr)(bp + 4), (Ptr)&ddpAddress,
                                              sizeof(DDPAddress));
        BlockMove((Ptr)(bp + alen + 4), (Ptr)nameString, nlen);
        nameString[nlen] = '\0';

        /* Print, display, or store the address
         and name in ddpAddress and nameString. */

        /* point to next tuple */
        bp = bp + ((len + 3L) & ~3L);
        index++;
    }
}
```

The `DoParseOTLookup` function takes two parameters, a pointer to the buffer containing the data returned by the `OTLookupName` function and a value specifying the number of entries in the buffer. Both these values are returned in the `reply` parameter to the `OTLookupName` function. The `DoParseOTLookup` function uses a `while` loop to move through the buffer entry by entry. For each entry,

- it determines the length of the address by looking at the first 2 bytes of the entry and determines the length of the name by looking at the next 2 bytes of the entry
- it sets `len` to the length of the entire entry by adding 4 bytes (the room taken up by `addrLen` and `nameLen`) to the length of the address and the length of the name
- it moves the DDP address, which it finds 4 bytes into the entry, into the `ddpAddress` variable; and it moves the NBP name, which starts at (`bp + alen + 4`) into the `nameString` variable.

  Because the NBP name is neither a Pascal nor a C string (it does not begin with a length byte and it does not end with a null character), the function then adds a null character to the name stored in the `nameString` variable to

make it a C string. This makes it easier for the program to manipulate the string.

At this point in the program, you can print or display or store the values of the `ddpAddress` and `nameString` variables. The statement

```
bp = bp + ((len + 3L) & ~3L);
```

is used to point to the next entry.

If you execute the `OTLookupName` function asynchronously, you can also use method described in the next section, "Retrieving Entries in Asynchronous Mode," to retrieve address-name information.

Retrieving Entries in Asynchronous Mode

If you call the `OTLookupName` function asynchronously, you can use an alternate method for retrieving matching entries. In asynchronous mode, this function returns two event codes: it returns the `T_LKUPNAMERESULT` code each time it stores a name in the reply buffer, and it returns the `T_LKUPNAMECOMPLETE` code when it has stored the last name in the reply buffer—that is, when the function as a whole completes execution. You can ignore the `T_LKUPNAMERESULT` event, allocate a large reply buffer, and use the method described in the previous section, "Retrieving Multiple Entries From the Reply Buffer," to parse through the buffer. Alternately, each time the `T_LKUPNAMERESULT` event is passed to your notification function, you can do the following:

1. Copy the name and address information from the reply buffer to some other location.
2. From inside the notifier function, set the `reply->names.len` field or the `reply->rspcount` field to 0.

   When you set either of these fields to 0, Open Transport automatically sets the other field to 0. It's important, however, that you reset these values from within the notifier or the results might be unpredictable.
3. Repeat the first two steps until the event passed to your notifier function is `T_LKUPNAMECOMPLETE`.

This method saves you the trouble of guessing how large a reply buffer to allocate. It might also save you some memory if you are expecting many matches to be returned and are interested in only some of them.

Mappers Reference

This section describes the data types and functions that you use with mappers. You can also use general provider data types and functions with mappers. General structures and functions are described in the reference section of the chapter "Providers" earlier in this book.

Constants and Data Types

This section describes the data types used by mapper functions.

The TRegisterRequest Structure

You use the `TRegisterRequest` structure to specify the entity name you want to register using the `OTRegisterName` function (page 4-22) and, optionally, to specify its address.

The `TRegisterRequest` structure is defined by the `TRegisterRequest` data type.

```
struct TRegisterRequest
    {   TNetbuf name;
        TNetbuf addr;
    };
    typedef struct TRegisterRequest TRegisterRequest;
```

Field descriptions

`name` A `TNetbuf` structure that specifies the location and size of a buffer containing the entity name you want to register. You must allocate a buffer that contains the name, set the `name.buf` field to point to that buffer, and set the `name.len` field to the length of the name.

`addr` A `TNetbuf` structure that specifies the location and size of a buffer containing the address associated with the entity whose name you want to register. You must allocate a buffer that contains the address, set the `addr.buf` field to point to that buffer, and set the `addr.len` field to the length

of the address. The actual address with which the entity is associated is returned in the `addr` field of the `TRegisterReply` structure.

You can set the `addr.len` field to 0, in which case the underlying protocol, finds an appropriate address to associate with the newly registered entity name.

The TRegisterReply Structure

You use the `TRegisterReply` structure to store information returned by the `OTRegisterName` function (page 4-22).

The `TRegisterReply` structure is defined by the `TRegisterReply` data type.

```
struct TRegisterReply
    {   TNetbuf     addr;
        OTNameID nameid;
    };
    typedef struct TRegisterReply TRegisterReply;
```

Field descriptions

`addr` A `TNetbuf` structure that you allocate to hold the location and size of a buffer containing the actual address of the entity whose name you have just registered. This information is passed back to you when the `OTRegisterName` function returns. You must allocate a buffer, set the `addr.buf` field to point to it, and set the `addr.maxlen` field to the maximum size the address could take up.

`nameid` A unique identifier passed to you when the `OTRegisterName` function returns. You can use this identifier when you call the `OTDeleteNameByID` function to delete the name.

The TLookupRequest Structure

You use the `TLookupRequest` structure to specify the registered entity name to be looked up by the `OTLookupName` function (page 4-26) and to set additional values that the mapper provider uses to circumscribe the search.

The `TLookupRequest` structure is defined by the `TLookupRequest` data type.

```
struct TLookupRequest
    {   TNetbuf     name;       /* name to search for */
        TNetbuf     addr;       /* address bound to named endpoint */
        UInt32      maxcnt;     /* how many matches are expected */
        OTTimeouttimeout;       /* how long to continue search */
    };
    typedef struct TLookupRequest TLookupRequest;
```

Field descriptions

`name` A `TNetbuf` structure specifying the location and size of a buffer that contains the name to be looked up. You must allocate a buffer that contains the name, set the `name.buf` field to point to it, and set the `name.len` field to the length of the name.

`addr` A `TNetbuf` structure describing the address of the node where you expect the names are stored. You should normally supply 0 for `addr.len`. This causes a protocol family like TCP/IP to use the address of the name server selected in the control panel as the destination of its search. For a protocol family like AppleTalk, in which every node has access to name and address information, this parameter is meaningless.

Specifying an address has meaning for those protocols that use a dedicated server or other device to store name information. In such a case, the name specified would override the protocol's default address. To specify an address, you would need to allocate a buffer containing the address, set the `addr.buf` field to point to it, and set the `addr.len` field to the length of the address. Consult the documentation supplied with your protocol to determine whether you can or should specify an address.

`maxcnt` A long specifying the number of names you expect to be returned. Some protocols allow the use of wildcard characters in specifying a name. As a result, the `OTLookupName` function might find multiple names matching the specified name pattern. If you expect a specific number of replies for a particular name, you should specify this number to obtain faster execution. There is no default value for this field.

`timeout`	A long specifying the amount of time, in milliseconds, that should elapse before Open Transport gives up searching for a name. The default value is 0.

The TLookupReply Structure

You use the `TLookupReply` structure to store information passed back to you by the `OTLookupName` function (page 4-26). The information includes both a pointer to a buffer that contains registered entity names matching the criterion specified with the `TLookupRequest` structure and the number of names found.

The `TLookupReply` structure is defined by the `TLookupReply` data type.

```
struct TLookupReply
    {   TNetbuf     names;
        UInt32      rspcount;
    };
    typedef struct TLookupReply TLookupReply;
```

Field descriptions

`names`	A `TNetbuf` structure that specifies the size and location of a buffer into which the `OTLookupName` function, on return, places the names it has found. You must allocate a buffer in which the replies are stored when the function returns; you must set the `names.buf` field to point to it; and you must set the `names.maxlen` field to the maximum size of the buffer.
`rspcount`	A long specifying the number of names found.

The TLookupBuffer Structure

A mapper provider uses the `TLookupBuffer` structure to be able to parse through the buffer passed back in the `reply` parameter to the `OTLookupName` function (page 4-26). When you allocate a buffer in which the `OTLookupName` function places the names it has found, you must cast it as a `TLookupBuffer` structure. You must make sure that the buffer you allocate is large enough to contain all the names returned by the `OTLookupName` function, plus up to 3 bytes of padding for each name, plus an additional 8 bytes for each name returned. Figure 4-1 on page 4-8 shows the structure of the reply buffer.

The `TLookupBuffer` structure is defined by the `TLookupBuffer` data type.

```
struct TLookupBuffer
    {   UInt16      fAddressLength;
        UInt16      fNameLength;
        UInt8       fAddressBuffer[1];
    };
```

Field descriptions

`fAddressLength`	Specifies the size of the address specified by the `fAddressBuffer` field.
`fNameLength`	Specifies the size of the name that is stored in the buffer following the `fAddressBuffer` field.
`fAddressBuffer`	Specifies the address to which the entity whose name follows (in the buffer) is bound.

Functions

This section describes mapper functions, provider functions that you use only with mappers to manage the mapping of entity names to endpoint addresses for a network. These functions fall into three categories: functions you use to create a mapper, functions you use to register a name or delete a registered name, and functions you use to search for a name or to validate a name-address pair.

As with other provider functions, you can execute mapper functions synchronously or asynchronously. Note, however, that Open Transport provides no function to cancel outstanding asynchronous mapper functions. The only way to cancel such functions is to close the mapper by calling the `OTCloseProvider` function, described in the chapter "Providers" earlier in this book.

You can also use general provider functions with mappers. You use these functions to change a function's mode of operation (for example, to blocking). General provider functions are described in the reference section of the chapter "Providers."

Creating Mappers

Before you can call mapper functions to register a name or search for a name, you must create a mapper provider by calling the `OTAsyncOpenMapper` or `OTOpenMapper` functions. When you finish using a mapper, call the `OTCloseProvider` function to close and delete the mapper provider.

OTAsyncOpenMapper

Creates a mapper and installs a notifier function for the mapper provider. The `OTAsyncOpenMapper` function is asynchronous and creates a mapper that operates asynchronously by default.

C INTERFACE

```
OSErr OTAsyncOpenMapper (OTConfiguration* config, OTOpenFlags oflag,
                         OTNotifyProcPtr proc, void* contextPtr);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`config`	A pointer to an endpoint configuration structure that specifies the mapper's characteristics. You obtain a value for the `config` parameter by calling the `OTCreateConfiguration` function. The `OTAsyncOpenMapper` function deletes the configuration structure after creating the mapper or attempting to create it.
`oflag`	Reserved; must be set to 0.
`proc`	A pointer to a notifier function for this mapper. If you do not provide a notifier function, your application cannot receive Open Transport events, including the event advising you that the mapper has been created.

`contextPtr` A context pointer for your use. The mapper provider passes this pointer value when calling the notifier function you specify in the `proc` parameter. You might use the `contextPtr` parameter, for example, to pass to your notifier function information about your application's current context.

DESCRIPTION

The `OTAsyncOpenMapper` function opens a mapper having the configuration specified by the `config` parameter. For additional information see the chapter "Configuration Management" and the documentation provided for the name-binding protocol you are using. The `OTAsyncOpenMapper` function runs asynchronously, returning a result code as soon as the function has been queued for execution.

The `OTAsyncOpenMapper` function attempts to create a mapper, and then calls the notifier function that you specified in the `proc` parameter, passing `T_OPENCOMPLETE` for the `code` parameter, a result code in the `result` parameter, and the mapper reference for the newly created mapper in the `cookie` parameter.

A mapper created by the `OTAsyncOpenMapper` function operates in asynchronous mode, unless you change the mapper's mode of execution by calling the `OTSetSynchronous` function. When a mapper is in asynchronous mode, all provider functions that use the mapper execute asynchronously.

By default, a newly created mapper does not block and does not acknowledge sends. To change the mapper's default mode of operation, you can call the `OTSetBlocking` function and the `OTIsAckingSends` function.

You can open multiple mappers using identical or different configurations, although if you use identical configurations, you must read the "Special Considerations" section. The different mappers can be distinguished by the mapper reference. You can set the `contextPtr` parameter to point to the mapper reference or to a structure containing the mapper reference; this allows your notifier function to determine to which mapper a completion event belongs.

SPECIAL CONSIDERATIONS

The `OTAsyncOpenMapper` function destroys the configuration value returned by the `OTCreateConfiguration` function. You cannot use the same configuration to open multiple mappers. To obtain a valid copy of the configuration for use

when opening another mapper, you must call the `OTCloneConfiguration` function.

COMPLETION EVENT CODES

`T_OPENCOMPLETE`	0x20000007	The `OTAsyncOpenMapper` function has completed. The `cookie` parameter of the notifier function points to the mapper reference for the new mapper.

SEE ALSO

To open a mapper in synchronous mode, use the `OTOpenMapper` function (page 4-19).

To close and delete a mapper use the `OTCloseProvider` function, described in the chapter "Providers" in this book.

For more information about a mapper's mode of operations, see the section "Setting Modes of Operation for Mappers" on page 4-5.

The `OTCreateConfiguration` function used to create the configuration structure that defines the protocols underlying the mapper is discussed in the chapter "Configuration Management" in this book.

The `OTSetAsynchronous` function, the `OTSetBlocking` function, the `OTIsAckingSends` function, and the notifier function are described in the chapter "Providers" in this book.

OTOpenMapper

Creates a mapper provider and returns a mapper reference. This function is synchronous and creates a mapper that operates synchronously.

C INTERFACE

```
MapperRef OTOpenMapper(OTConfiguration* config, OTOpenFlags oflag,
                       OSErr* err)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`config`	A pointer to a configuration structure that specifies the mapper's characteristics. You obtain a value for the `config` parameter by calling the `OTCreateConfiguration` function. The `OTOpenMapper` function deletes the configuration structure when creating the mapper or attempting to create it.
`oflag`	Reserved; must be set to 0.
`err`	A pointer to the result code for this function.

DESCRIPTION

The `OTOpenMapper` function opens a mapper having the configuration specified by the `config` parameter. For additional information see the chapter "Configuration Management" and the documentation provided for the name-binding protocol you are using. The function returns a mapper reference, by which you refer to the created mapper when calling mapper functions. If the `OTOpenMapper` function fails, its return value is `NULL`.

A mapper created by the `OTOpenMapper` function operates in synchronous mode, unless you change the mapper's mode of execution by calling the `OTSetAsynchronous` function. When a mapper is in synchronous mode, all mapper provider functions execute synchronously.

By default, a newly created mapper does not block and does not acknowledge sends. To change the mapper's default mode of operation, you can call the `OTSetBlocking` function and the `OTIsAckingSends` function.

You can open multiple mappers using identical or different configurations, although if you use identical configurations, you must read the "Special Considerations" section, next. The different mappers can be distinguished by the mapper reference.

SPECIAL CONSIDERATIONS

Because the `OTOpenMapper` function executes synchronously, your application should not call this function at interrupt time.

The `OTOpenMapper` function destroys the configuration structure returned by the `OTCreateConfiguration` function. If you want to use the same configuration to open additional mappers, you must obtain a valid copy of the configuration structure by calling the `OTCloneConfiguration` function.

SEE ALSO

The `OTCreateConfiguration` function used to create the configuration structure that defines the protocols underlying the mapper is discussed in the chapter "Configuration Management" in this book.

To create a mapper asynchronously, call the `OTAsyncOpenMapper` function (page 4-17).

To close and delete a mapper, call the `OTCloseProvider` function, described in the chapter "Providers" in this book.

For additional information about a mapper's mode of operations, see "Setting Modes of Operation for Mappers" on page 4-5.

The `OTSetAsynchronous` function, the `OTSetBlocking` function, and the `OTIsAckingSends` function are described in the chapter "Providers" in this book.

Registering and Deleting Names

You use the mapper functions described in this section to register a name on the network and to delete a name from the network.

OTRegisterName

Registers an entity name on the network.

C INTERFACE

```
OSErr OTRegisterName (MapperRef ref, TRegisterRequest* request,
                      TRegisterReply* reply);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`ref`	A mapper reference.
`request`	A pointer to a `TRegisterRequest` structure (page 4-12) that specifies the entity name you want to register and the endpoint address.
`reply`	A pointer to a `TRegisterReply` structure (page 4-13) that specifies the address and ID of the endpoint whose name is being registered.

DESCRIPTION

If the name-registration protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` function supports dynamic name and address registration, you can use the `OTRegisterName` function to make a name visible on the network to other network devices.

Some protocol implementations under Open Transport allow a client to specify a name rather than an address when binding the endpoint using the `OTBind` function. Binding an endpoint by name causes the protocol to automatically register the name on the network if the protocol supports dynamic name registration. This is the simpler technique for registering a name and is preferred over creating a mapper provider and then using the `OTRegisterName` function to register the name.

The format for the requested name and address is specific to the protocol used. Please consult the documentation for the protocol you are using for format information.

COMPLETION EVENT CODES

T_REGNAMECOMPLETE	0x2000000D	The OTRegisterName function has completed. The cookie parameter of the notifier function points to the reply parameter.

SEE ALSO

You use the OTLookupName function (page 4-26), to search for a registered name or to confirm that a name has been registered.

You use the OTDeleteName function (described next) or the OTDeleteNameByID function (page 4-25) to remove a previously registered name.

You use the OTOpenMapper function (page 4-19) or OTAsyncOpenMapper function (page 4-17) to create a mapper.

The OTBind function is described in the chapter "Endpoints" in this book.

For information on how to use this function with a TCP/IP protocol, see page 8-20 in the TCP/IP chapter.

Notifier functions are described in the chapter "Providers" in this book.

OTDeleteName

Removes a previously registered entity name.

C INTERFACE

```
OSErr OTDeleteName (MapperRef ref, TNetbuf* name);
```

C++ INTERFACES

```
TMapper::DeleteName(TNetbuf* name);
```

PARAMETERS

`ref`	The mapper reference of the mapper you are using to delete the name.
`name`	A `TNetbuf` structure describing the name to be removed. You must allocate a buffer that contains the name, set the `name.buf` field to point to the buffer, and set the `name.len` field to the length of the name.

DESCRIPTION

If the name-registration protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` function supports dynamic name and address registration, you can use the `OTDeleteName` function to delete a registered name.

COMPLETION EVENT CODES

`T_DELNAMECOMPLETE`	0x2000000E	The `OTDeleteName` function has completed. The `cookie` parameter of the notifier function points to the `name` parameter.

SEE ALSO

The `OTRegisterName` function you used to register the name returns an ID value for the registered name in its `reply` parameter. You might find it more convenient to use the `OTDeleteNameByID` function (described next) to delete a name using this ID value than to use the `OTDeleteName` function.

For information on how to use this function with a TCP/IP protocol, see page 8-20 in the TCP/IP chapter.

You use the `OTOpenMapper` function (page 4-19) or `OTAsyncOpenMapper` function (page 4-17) to create a mapper.

OTDeleteNameByID

Removes a previously registered name as specified by its name ID.

C INTERFACE

```
OSErr OTDeleteNameByID (MapperRef ref, OTNameID id);
```

C++ INTERFACES

```
TMapper::DeleteName(OTNameID id);
```

PARAMETERS

`ref`	A mapper reference.
`id`	The name ID, a long specifying a number that identifies the registered name.

DESCRIPTION

If the name-registration protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` function supports dynamic name and address registration, you can use the `OTDeleteNameByID` function to delete a registered name.

If the mapper is in asynchronous mode, the `OTDeleteNameByID` function returns immediately. When the function completes execution, the mapper provider calls the notifier function, passing `T_DELNAMECOMPLETE` for the `code` parameter, and a pointer to the `id` parameter in the `cookie` parameter.

SEE ALSO

The name ID that you delete using the `OTDeleteNameByID` function is returned in the `reply` parameter to the `OTRegisterName` function (page 4-22).

You use the `OTOpenMapper` function (page 4-19) or `OTAsyncOpenMapper` function (page 4-17) to create a mapper.

Looking Up Names

You use the `OTlookUpName` function to look up an entity name, to search for all names matching a specified pattern, or to confirm that a name is registered.

OTLookupName

Finds and returns all addresses that correspond to a particular name or name pattern, or confirms that a name is registered.

C INTERFACE

```
OSErr OTLookupName  (MapperRef ref, TLookupRequest* req,
                     TLookupReply* reply);
```

C++ INTERFACES

```
OSErr TMapper::LookupName(TLookupRequest* req,
                          TLookupReply* reply);
```

PARAMETERS

`ref`	A mapper reference.
`req`	A `TLookupRequest` structure (page 4-13) that specifies the name to be looked up as well as some additional values that the mapper provider can use to circumscribe the search.
`reply`	A `TLookupReply` structure (page 4-15) that specifies the size and location of a buffer containing the names found, and the number of names found.

DESCRIPTION

You can use the `OTLookupName` function to find out whether a name is registered and what address is associated with that name. You use the `req` parameter to supply the information needed for the search: what name should be looked up

and, optionally, what node contains that information, how many matches you expect to find, and how long the search should continue before the function returns. On return, the `reply` parameter contains the `names` field that points to the buffer where the matching entries are stored and the `rspcount` field that specifies the number of matching entries.

For each registered name found, the `OTLookupName` function stores the following information in the buffer referenced by the `names` field of the `reply` parameter:

```
unsigned short addrLen;         /* length of address that follows*/
unsigned short nameLen;         /* length of name that follows */
unsigned char addr[];           /* address */
unsigned char name[];           /* name, padded to quad-word boundary*/
```

If you are searching for names using a name pattern and you expect that more than one name will be returned to you, you need to parse the reply buffer to extract the matching names.

If you call the `OTLookupName` function asynchronously, the mapper provider calls your notifier function passing one of two completion codes for the `code` parameter (`T_LKUPNAMERESULT` or `T_LKUPNAMECOMPLETE`) and passing the `reply` parameter in the `cookie` parameter. The mapper provider passes the `T_LKUPNAMERESULT` code each time it stores a name in the reply buffer, and it passes the `T_LKUPNAMECOMPLETE` code when it is done. When you receive this event, examine the `rspcount` field to determine whether there is a last name to retrieve from the reply buffer. The use of both codes is a feature that gives you a choice about how to process multiple names when searching for names matching a pattern.

- If you decide to allocate a buffer that is large enough to contain all the names returned, you can ignore the `T_LKUPNAMERESULT` code and call a function that parses the buffer once the `OTLookupName` function has completed—that is, once the provider calls your notifier function using the `T_LKUPNAMECOMPLETE` event.
- If you want to save memory or if you don't know how large a buffer to allocate, you can use the following method to process the names returned. Each time that the `T_LKUPNAMERESULT` event is passed, you must do something with the reply from the reply buffer. You can copy it somewhere, or you can delete it if it isn't a name you're interested in. Then, from inside your notifier you must set the `reply->names.len` field or the `reply->rspcount` field back to 0 (thus allowing the mapper provider to overwrite the original name). This tells the mapper provider that you are ready to receive another name.

Accordingly, when the mapper provider has inserted another name into your reply buffer, it calls your notifier passing the `T_LKUPNAMERESULT` code, and you can process the new entry as you have processed the first entry. This method also saves you the trouble of having to parse through the buffer to extract name and address information.

The `cookie` parameter to the notifier contains the `reply` parameter.

The format of the names and protocol addresses are specific to the underlying protocol. Consult the documentation supplied for your protocol for more information.

COMPLETION EVENT CODES

`T_LKUPNAMECOMPLETE`	0x2000000F	The `OTLookupName` function has completed. The `cookie` parameter of the notifier function points to the `reply` parameter.

SEE ALSO

You use the `OTDeleteName` function (page 4-23) or the `OTDeleteNameByID` function, (page 4-25) to delete a registered name.

A sample program that parses the reply buffer to extract matching names is shown in the section "Searching for Names," beginning on page 4-7.

For information on how to use this function with a TCP/IP protocol, see page 8-20 in the TCP/IP chapter.

Notifier functions are described in the chapter "Providers" in this book.

Option Management

Contents

This chapter explains the use of options, values associated with an endpoint provider, which you can change to fine-tune or customize the data-transfer service offered by the endpoint. In general, the use of options decreases portability and makes transport independence much more difficult, if not impossible, to achieve. Therefore, it is important to note that default option values are provided for every type of endpoint and that you can write applications that never need to specify any options. You need to read this chapter if

- you need to use services that must be specified using options

  For example, you are using a transaction-based endpoint and need to be able to send expedited data in order to forward an attention message.

- it is critical to your application that you fine-tune the data-transfer services offered by a protocol and you can only do this by using options

  For example, you need to manipulate the size of internal send and receive buffers to eliminate data backlog or buffer overflow problems.

- you need to create a debugging version of the application through the use of options

This chapter describes general options that can be specified by any protocol that supports them, explains the rules followed in the negotiation process, and explains how you construct an options buffer and how you get and set option values. It also describes functions that you can use

- to construct buffers containing option information
- to locate options in these buffers
- to parse buffers containing option information

To understand this chapter, you should be familiar with endpoint providers and the endpoint functions used to transfer data. These topics are discussed in the "Endpoints" chapter in this book. For specific information about the options that are supported for a protocol implementation, you need to consult the documentation provided for that protocol.

About Options and Option Negotiation

For every endpoint, Open Transport maintains an options buffer. When you create an endpoint provider, Open Transport fills this buffer with a default value for each option supported for the endpoint. Option values have meaning for and are defined by the protocol to which they apply. Typically, Open Transport uses endpoint options to control aspects of the endpoint's operation. For example, if a protocol guarantees reliable delivery of data, the protocol might define an option that specifies the number of times a send operation is retried before the send fails and an error message is generated. Protocol implementations provide default values for options to ensure maximum portability for your application across protocol families and system platforms.

In writing a networking application, you can use an endpoint provider's default option values or you can replace these with other values to control the behavior of an endpoint. **Option negotiation** describes the process that results when you decide to replace default values with option values that you choose. A successful negotiation results in your obtaining exactly the option values you requested, a partly successful negotiation results in your getting different values for the options you requested, and a failed negotiation results in your not being able to change existing values at all.

Depending on the option you want to modify, a negotiation might involve a client and its endpoint provider, or it might involve both a local and remote client and their endpoint providers. In either case, it's important to keep in mind that the process is a negotiation—that is, before you can change the characteristics of an endpoint or change the way in which it transfers data or establishes a connection, an agreement has to be reached. If you cannot reach this agreement, the operation you are attempting to complete could fail. In this case, you might have to find a way of implementing the service you need, other than through the use of options.

Explicit Use of Options and Portability of Code

The goal of the Open Transport architecture is to enable networking applications to migrate across protocol families and system platforms with little or no change to code. However, the price of transport independence or, ideally, transport transparency is that an application must be ready to forego

features that are unique to a specific protocol in order to work equally well with protocols offering a similar type of service, such as connection-oriented transactionless service or connectionless transaction-based service. Because options are often coupled with a particular protocol or protocol family, making explicit use of options degrades portability across protocol families. Similarly, different system platforms might offer different option support for the same protocols due to different implementations. Thus, making use of options can also endanger portability across different system platforms.

Note, however, that protocols are not necessarily interchangeable and that you might very reasonably want to take advantage of a protocol feature that is only available through the use of options. If this is the case, you need to become familiar with the material presented in the following sections, which describe the Open Transport rules for option management and negotiation. These rules have been defined to allow as much flexibility as possible so that even once an application chooses to make explicit use of options, it is still possible to negotiate a compromise that is acceptable to all involved parties. In this sense, the most important thing to understand about most options is that each value is not fixed but always negotiated relative to the context within which the endpoint provider operates. For this purpose, context might include the protocol implementation, the state of the endpoint, and current option values.

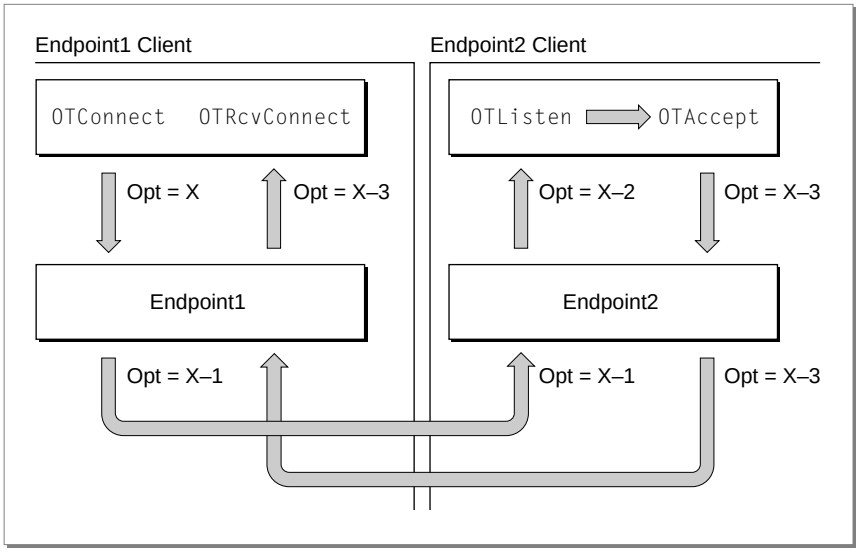
Types of Options

Options can be association-related, privileged, read-only, or absolute.

Association-related options are specified in relation to a particular connection, data transmission, or transaction; such options include information that is destined for the remote client. The client initiating the connection or transaction, or sending the datagram, initially defines the value of an association-related option; but the endpoint providers and the remote client can also negotiate this value (almost always to a less-desirable value). Figure 5-1 illustrates the extreme case, in which each agent involved in the process of establishing a connection renegotiates an association-related option proposed by the active peer. When the client application calls the `OTConnect` function, it specifies some value x for an option. The endpoint provider, Endpoint1, lowers this value before passing it to the remote endpoint, Endpoint2. The remote endpoint lowers the value further before notifying its client of the incoming connection. When the `OTListen` function returns, it specifies the option value X–2. The remote client decides to accept the connection using the `OTAccept` function but also to lower it further to X–3.

When the client that initiated the connection receives the remote client's response via the `OTRcvConnect` function, it can examine the option values to determine the final negotiated value for the option it requested. (By way of example, Figure 5-1 shows that the negotiated value is lowered at each stage of the negotiation. Depending on the option being negotiated, however, a higher value could result from the degradation resulting from a negotiation.)

Figure 5-1 Negotiating an association-related option



By contrast, options that are **non-association–related** are negotiated solely between a client application and an endpoint provider. Such options do not contain information that involve the remote client. For example, the client application can specify an option that permits debugging or that increases the size of an internal receive buffer. Table 5-1 shows which Open Transport functions can accept association-related options and which can accept both types of options for input and output parameters that you can use to specify options.

Table 5-1 Open Transport endpoint functions and the types of options they accept

Function	Input parameter	Output parameter
`OTListen`	Not applicable	Association-related
`OTRcvUData`	Not applicable	Association-related
`OTRcvURequest`	Not applicable	Association-related
`OTRcvConnect`	Not applicable	Both
`OTRcvUDErr`	Not applicable	Both
`OTAccept`	Both	Not applicable
`OTSndUData`	Both	Not applicable
`OTSndURequest`	Both	Not applicable
`OTConnect`	Both	Not applicable
`OTOptionManagement`	Both	Not applicable

Privileged options are options or option values that you can only set or change if you are a privileged client. The fact that an option is privileged affects the outcome of option negotiation if a nonprivileged client attempts to set such an option. In some cases, nonprivileged clients can read the value of a privileged option.

Read-only options, as the name implies, are options whose values you can read but not change. For example, a protocol implementation might determine that a client cannot change the maximum length of a transport data unit; nevertheless, it would be important that the client be able to find out what the maximum length is in order to set up sufficiently large buffers for incoming data.

Whether an option is read-only depends on the status of the client and on the state of the endpoint. Depending on the implementation, an option might be

- read-only for all clients or just for nonprivileged clients
- negotiable in some endpoint states and read-only in other states

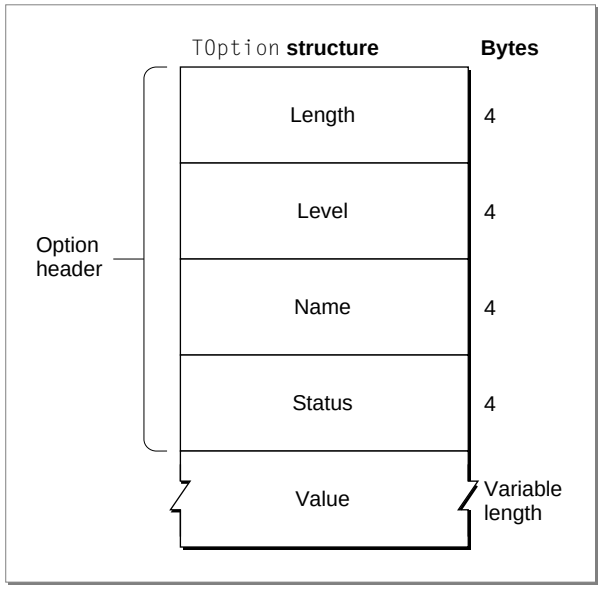
  For example, for TCP/IP endpoints, the ISO quality-of-service options are negotiable when the endpoint is in the `T_IDLE` and `T_INCON` states, and read-only in all other states except `T_UNINIT`.

Options that are **absolute requirements** are options that a protocol must implement. This means that a protocol can neither ignore such an option nor negotiate it to a lower value. (Options that are not absolute requirements can be negotiated to a lower value, in which case the negotiation is deemed to be partly successful.) If the proposed option is an absolute requirement and the negotiated value is not the same as the proposed value, the negotiation fails, and any attempt to establish a connection or to send data also fails.

The Format of Option Information

An option has a name and a value, it is defined for a specific protocol, and it takes up a certain amount of room in memory. The `TOption` structure used to define an option contains fields for each of these characteristics. As Figure 5-2 shows, an option is described by an option header and a value.

Figure 5-2 The format of option information



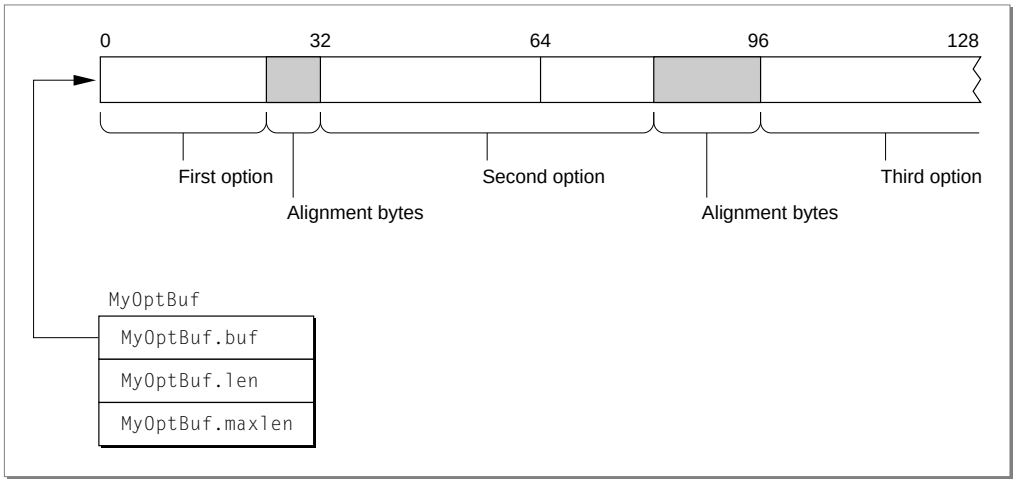
The option header is the same for all options. It contains four fields that specify:

- The length of the entire structure. The length includes the length of the option header and the length of the value field; it does not include added padding.
- The protocol (level) for which the option applies. It is possible to set an option for any protocol that is part of an endpoint provider's configuration. For example, if you open an AppleTalk Transaction Protocol (ATP) endpoint, it is possible to set an option at the Datagram Delivery Protocol (DDP) level by specifying DDP for the `level` field.
- The name of the option. Each protocol implementation defines the names of options it supports.
- The status of the option. The endpoint provider fills in this field to indicate the outcome of the option negotiation.

The length and format of data in the value field depend on the option being defined.

You store option information for an endpoint in a buffer containing one or more `TOption` structures. A `TNetbuf` structure describes the buffer. Figure 5-3 shows a `TNetbuf` structure, `MyOptBuf`, that describes an options buffer containing three options. The field `MyOptBuf.buf` points to the buffer; the field `MyOptBuf.len` specifies the actual length of the buffer.

Figure 5-3 An options buffer



You can concatenate several `TOption` structures in a buffer, as shown in Figure 5-3, provided you observe the following rules:

- `TOption` structures must be quad-word aligned within the buffer.
- If you are using the `OTOptionManagement` function to set or verify option values, all options in the buffer must be for the same protocol. That is, the value of the `level` field must be the same. When used with any other function, the options buffer can contain options set for different protocols.

XTI-Level Options and General Options

In addition to options defined for specific protocols, Open Transport defines options called *XTI-level options* that are not specific to a particular endpoint. Some of these options are absolute requirements, which means that whatever protocol you are using must support these options. You need to consult the documentation for your protocol to determine the meaning of the option for your endpoint and for additional information about default values and ranges or valid values supported for the option. Table 5-2 provides a brief summary of XTI-level options. For more detailed information about these options, see "XTI-Level Options" on page 5-25.

Table 5-2 XTI-level options

Option name	Description
`XTI_DEBUG`	Enables debugging
`XTI_LINGER`	Specifies a linger period which delays the execution of the `OTCloseProvider` function
`XTI_RCVBUF`	Specifies the size of your endpoint's internal receive buffer
`XTI_RCVLOWAT`	Specifies the minimum number of bytes that can accumulate in the endpoint's internal receive buffer before your application receives a `T_DATA` event signalling the arrival of data
`XTI_SNDBUF`	Specifies the size of your endpoint's internal send buffer
`XTI_SNDLOWAT`	Specifies the minimum number of bytes that can accumulate in the endpoint's internal send buffer before the provider actually sends the data

In addition to the XTI-level options, Open Transport defines the set of generic options listed in Table 5-3. None of these options are absolute requirements. This means that if an Open Transport protocol supports the functionality of one of these options, it should use this option to do it. For additional information about generic options, see "Generic Options" on page 5-28.

Table 5-3 Open Transport generic options

Option name	Description
OPT_CHECKSUM	Specifies whether packets have checksums calculated on receipt
OPT_RETRYCNT	Specifies the number of times a function can attempt packet delivery
OPT_INTERVAL	Specifies the amount of time to wait between attempts to deliver a packet or request
OPT_ENABLEEOM	Specifies whether the T_MORE flag for the OTSnd function can be used to signal the end of a logical unit
OPT_SELFSEND	Specifies whether self-sending is enabled for broadcast messages.
OPT_SERVERSTATUS	Specifies the status string that is used to answer a SendStatus request from a client.
OPT_KEEPALIVE	Specifies the amount of time a connection should be maintained in the absence of data transfer

Using Options

This section describes the rules for option negotiation and how negotiation is affected by the function you use to set options. It also explains how you use endpoint functions to set and retrieve option values and how you use Open Transport utility functions to construct an options buffer and parse through an options buffer.

If your application needs to negotiate option values, you must read the sections "Determining Which Function to Use to Negotiate Options," "Negotiating Options," and "Obtaining the Maximum Size of an Options Buffer." After reading these sections, you can read whichever of the remaining sections describes the task you need to accomplish.

Determining Which Function to Use to Negotiate Options

You can negotiate options using the `OTOptionManagement` function or using any one of the endpoint functions used to transfer data or establish a connection. The following bulleted list summarizes the major differences between using the `OTOptionManagement` function or using other endpoint functions to set an option value.

- Options specified using the `OTOptionManagement` function affect all functions called by an endpoint. Options specified using individual endpoint functions affect only the connection, transaction, or datagram for which they are set. For example, you can call the `OTOptionManagement` function to turn the checksum option on; you could override that value by calling the `OTSndUData` function and turning the checksum option off for the duration of that function call. The next time you call the `OTSndUData` function, the default value, set with the `OTOptionManagement` function would apply, so the checksum option would be off.
- The `OTOptionManagement` function is the only way that you can obtain default option values or check for current values of all options supported by an endpoint.
- When attempting to set multiple options, if an option is illegal or rejected, the `OTOptionManagement` function still returns successfully, indicating for each option in the buffer whether it has been successfully negotiated. In the same circumstances, any other function returns an error, and even though some of the options might have been successfully negotiated, you have no way of knowing which were and which were not.
- If you are using the `OTOptionManagement` function to set or verify option values, all options in the buffer must be for the same protocol. If you use any other function to negotiate options or to check their value, the buffer can contain options set for different protocols.
- If association-related options contain information that is transmitted across the network or if they affect the transmission itself, they take effect when Open Transport establishes the connection, sends the transaction, or

transmits the datagram. If you use the `OTOptionManagement` function to change such an option, the endpoint provider checks whether the option is supported and negotiates a value according to its current knowledge. Then it writes the negotiated value to the endpoint's internal options buffer. However, more negotiations might take place when the connection is established or the transaction or datagram is sent. This can result in a degradation of the option value or even in a negotiation failure. If the negotiation succeeds, the newly negotiated values are written to the internal options buffer.

Negotiating Options

This section describes the rules governing option negotiation and the error conditions that might occur during this process. Unless stated otherwise, these rules apply to all functions that allow you to specify option values.

A basic rule to keep in mind is that options change only as the result of successful negotiations or partly successful negotiations. If you use any function except the `OTOptionManagement` function, the changes last for the duration of that function invocation. Option values are not changed by a change in the state of an endpoint. Once you change an option value permanently, there is no function that you can call to restore an option to its previous value, unless that previous value is the default value.

Negotiating Multiple Options

You can use one function to negotiate several options by placing the options in the options buffer passed to the function. If one of the options is ignored or rejected for any reason, the outcome depends on the function you use to set options.

- If you use the `OTOptionManagement` function, the function returns the result of negotiating each option in the `status` field of each option. The failure of one or more options does not cause the function to fail.
- The `OTConnect`, `OTAccept`, `OTSndUdata`, or `OTSndURequest` functions might succeed or fail, depending on the implementation and on the error condition. Options that are not supported are generally ignored; they do not cause a function to fail or a connection to abort. However, if the endpoint provider is unable to negotiate options that are absolute requirements or options that are read-only, these functions will fail.

If option negotiation causes one of these functions to fail, it is possible that some options were successfully negotiated before the failure. However, it is not possible to determine which of the options caused the failure.

If you specify the same option more than once, the endpoint provider does not check for duplicate occurrences of the same option. It simply processes the options one after another. However, the endpoint provider might negotiate options in any order; therefore, it is not safe to make any assumptions that a later occurrence of an option will override an earlier occurrence.

Initiating an Option Negotiation

You initiate an option negotiation by calling the `OTOptionManagement` function with the flag `T_NEGOTIATE` set or by calling the `OTConnect`, `OTSndUData`, or `OTSndURequest` function and specifying an options buffer length that is greater than 0. You can specify values for some or all of the options supported by an endpoint. The endpoint provider takes values for options that you do not specify explicitly in the options buffer, from the endpoint's internal options buffer. This buffer contains the endpoint's current option values; these could be default values, values that you specified when you configured the provider, or values resulting from a previous negotiation.

If the endpoint supports an option, the possible outcome of option negotiation depends on whether the option is an absolute requirement, as described in the next two sections. If the endpoint does not support the option, the `OTOptionManagement` function reports `T_NOTSUPPORT` in the `status` field. The `OTConnect`, `OTSndUData`, or `OTSndURequest` functions ignore the option.

Options That Are Absolute Requirements

If the option is an absolute requirement, the result of the negotiation depends on whether the negotiated value is the same as the requested value. If it is, the `status` field in the `TOption` structure describing the option is set to `T_SUCCESS` when the function returns. If the negotiated value is not the same as the requested value, the result depends on the function used to negotiate the option:

- The `OTOptionManagement` function returns successfully, but the returned option has its `status` field set to `T_FAILURE`.
- A call to the `OTConnect` function fails. If the call is synchronous, the function returns with the `kOTLookErr` result. If the call is asynchronous, the endpoint

provider issues a T_DISCONNECT event to let you know that the connection has been rejected.

- The OTSndUData function fails with the kOTLookErr result; or if it returns successfully, the endpoint provider issues a T_UDERR event to indicate that the datagram was not sent.

Options That Are not Absolute Requirements

If the requested option is not an absolute requirement, the result of the negotiation depends on whether the negotiated value is the same as the requested value. If it is, the endpoint provider sets the status field of the TOption structure describing the option to T_SUCCESS. If the negotiated value is different than the proposed value, the endpoint provider sets the status field of the TOption structure describing the options to T_PARTSUCCESS.

Conflicting Option Values

It is possible that a requested option value conflicts with the value of another option that is proposed with the same call to the function or that is currently effective. The endpoint provider might not detect these conflicts immediately, and later they might lead to unpredictable results. If the endpoint provider detects conflicts at negotiation time, the conflicts are resolved according to the rules stated above.

An endpoint provider usually detects conflicts at the time it establishes a connection or sends a datagram. Consequently, if you use the OTOptionManagement function to set options, you might not become aware that there is a problem due to conflicting options until the options are actually exercised during connection establishment or data transmission.

Privileged or Read-Only Options

A protocol implementation can define options to be privileged or read only. These two categories are not necessarily separate. A privileged option might be inaccessible or read-only for nonprivileged clients. An option might be read-only for all clients or solely for nonprivileged clients. Here are two general guidelines to keep in mind:

- A client must be privileged to be able to change a privileged option.

  In the Macintosh implementation of Open Transport, there are no privileged options.

- A client cannot usually change the value of a read-only option.

  An option might be read-only in some endpoint states but not in others. 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`. Consult the documentation provided for the protocol you are using to determine whether an endpoint's state affects the status of read-only options.

If you request negotiation of a privileged option using the `OTOptionManagement` function, the function returns successfully with the `status` field of the privileged option set to `T_NOTSUPPORT`. If you use the `OTConnect`, `OTAccept`, `OTSndUData`, or `OTSndURequest` functions, the option is ignored—that is, the function result is not affected by the fact that the options are not supported.

If you request negotiation of a read-only option using the `OTOptionManagement` function, the function returns with the `status` field of the read-only option set to `T_READONLY`. If you use any other function to change a read-only option, the results vary with the function used:

- The `OTAccept` or `OTConnect` functions fail with the `kOTAccessErr` result, or the connection establishment aborts and the endpoint provider issues a `T_DISCONNECT` event. If the connection aborts, a synchronous call to `OTConnect` fails with the `KOTLookErr` result. Timing and the protocol implementation determine whether the `OTAccept` function succeeds or fails with the `kOTLookErr` result.
- The `OTSndUData` function might return the `kOTLookErr` result or return successfully, but the endpoint provider issues a `T_UDERR` event to indicate that it did not send the datagram.

Error Conditions

Option negotiation might be affected if you try to negotiate an illegal option, a privileged or read-only option, an unsupported option, or an option for an unsupported protocol (level). The results of attempting to negotiate privileged or read-only options are described in "Privileged or Read-Only Options" on page 5-15. This section explains the outcome of negotiating illegal options and describes other problems that might arise during option negotiation.

An option is illegal in these cases:

- It is the last option in an options buffer, and the length specified in the `TOption.len` field exceeds the remaining size of the options buffer. (The length of the option includes the option header as well as the option value.

See Figure 5-2 on page 5-8 for information about the format of option information in an options buffer.)

- The option value does not fall within the range of legal values for the option. The range of option values that are valid for a protocol implementation are given in the documentation provided for the protocol.

If you specify an illegal option, the following error conditions result depending on the function you used:

- The `OTOptionManagement` function returns with the `kOTBadOptionErr` result.
- Either the `OTAccept` or `OTConnect` function fails with a `kOTBadOptionErr` result, or the connection establishment aborts, depending upon the implementation and the time the illegal option is detected. If the connection aborts, the endpoint provider issues a `T_DISCONNECT` event. If `OTConnect` is executing synchronously, it fails with the `kOTLookErr` result. The `OTAccept` function either succeeds, or fails with the `kOTLookErr` result, depending on the implementation.
- The `OTSndUData` function fails with the `kOTBadOptionErr` result, or it returns successfully, but the endpoint provider issues a `T_UDERR` event to indicate that it did not sent the datagram.

If the options buffer you pass to a function contains multiple options and one of them is illegal, the function fails as described. However, if you used the `OTOptionManagement` function to set options, it is possible that some or all of the legal options in the buffer were successfully negotiated. You can check the current status for the endpoint by calling the `OTOptionManagement` function with the `T_CURRENT` flag set.

The `OTOptionManagement` function fails with the `kOTBadOptionErr` result if you specify an unknown value for the option protocol level. Using any other function to specify an unknown option level does not cause the function to fail, but results in the option being ignored.

Specifying an option name that is unknown or unsupported by the endpoint does not cause a function to fail. The `OTOptionManagement` function returns `T_NOTSUPPORT` in the `status` field for the option; the other endpoint functions ignore the unknown options.

Obtaining the Maximum Size of an Options Buffer

Different types of endpoints support different numbers of options. For example, an ATP endpoint might support more options than a DDP endpoint and might need a larger buffer to hold the options. When you call the `OTOptionManagement` function to change option values, the function returns in the `ret` parameter a pointer to the buffer containing the negotiated option values. You must have allocated the buffer used to store these options before calling the function. Likewise, when you call the `OTListen`, `OTRcvUData`, `OTRcvURequest` or `OTRcvConnect` functions, you can allocate a buffer in which current option values are to be placed when these functions return. In either case, you must specify the size of the buffer, and the buffer must be large enough to hold all of the endpoint's options. Otherwise, the function fails with a `kOTBufferOverflow` result. You can obtain the maximum size of a buffer used to store options for your endpoint by examining the `options` field of the `TEndpointInfo` structure for the endpoint. You can get a pointer to this structure when you open the endpoint, when you bind the endpoint, or when you call the `OTGetEndpointInfo` function.

Setting Option Values

You can use the `OTOptionManagement`, `OTAccept`, `OTSndUData`, `OTSndURequest`, and `OTConnect` functions to set option values. Setting option values results in a negotiation process between you (the client application) and the endpoint provider or, in the case of association-related options, between local and remote clients and their endpoint providers. The section "Initiating an Option Negotiation" on page 5-14 describes the rules that govern an option negotiation that you have initiated using the `OTOptionManagement`, `OTConnect`, `OTSndUData`, or `OTSndURequest` functions. The section "Retrieving Values for Connection-Oriented Endpoints," beginning on page 5-22 describes the negotiation rules that hold when you use the `OTOptionManagement` or `OTAccept` functions to respond to a negotiation. This section describes ways in which you can build the options buffer used to specify the options you want to change.

Specifying Option Values

No matter which function you use to set option values, you must allocate a buffer that contains the option value or values you want to change. The options in this buffer are described by `TOption` structures; the format of this structure is illustrated in Figure 5-2 on page 5-8. You can concatenate several structures in the buffer, as shown by Figure 5-3 on page 5-9, so long as each structure begins

on a long-word boundary. The buffer itself is described by a `TNetbuf` structure that specifies the location of the buffer and its size.

You can create a buffer that contains the option values you want to set in one of two ways: manually or by using the `OTCreateOptions` function. If you construct the buffer manually, you must do the following:

1. Allocate the buffer.
2. Create a `TOption` structure for each option you want to change.
3. Initialize each field of the `TOption` structure except for the `status` field.
4. Place the `TOption` structures in the buffer, making sure that each begins on a long-word boundary. This enables Open Transport to parse the buffer.
5. Append a null character to the end of the buffer. This enables Open Transport to tell that it has reached the end of the buffer.

To have Open Transport create a buffer for you, you must call the `OTCreateOptions` function and pass it a string containing one or more option values. This method saves time and trouble, but you can only use it if all the options in the buffer are for the same level and that level is the same as the top-level protocol for the endpoint provider. That is to say, you could not use this method to construct a buffer that contains DDP-level options for an ATP endpoint. In addition, this method is only guaranteed to work if you are building an options buffer for the `OTOptionManagement` function.

Listing 5-1 shows how you construct an options buffer manually. The listing creates and initializes two `TOption` structures, `ddpOpt` and `atpOpt`. It allocates a buffer large enough to contain the `TOption` structures and then places those structures in the buffer. Note that the structures are quad-word aligned and that a null character is appended to the end of the buffer.

Listing 5-1 Constructing an options buffer manually

```
TOption *ddpOpt, *atpOpt;
unsigned char optionBuffer[41];

ddpOpt = (TOption*)&optionBuffer[0];
ddpOpt->len = 20;
ddpOpt->level = ATK_DDP;
ddpOpt->name = OPT_CHECKSUM;
ddpOpt->status = 0;
```

```
ddpOpt->value[0] = 1;                        /* turn checksumming on */

atpOpt = (TOption*)&optionBuffer[20]
atpOpt->len = 20;
atpOpt->level = ATK_ATP;
atpOpt->name = OPT_RELTIMER;
atpOpt->status = 0;
atpOpt->value[0] = 2;       /* purge transaction list every 2 minutes */

optionBuffer[40] = 0;/* add null character to end of buffer */
```

Listing 5-2 shows how you construct an options buffer by using the `OTCreateOptions` function. The code initializes a string array, `myStr`, to hold option values. It then creates a `TOptMgmt` structure, which would later be passed to the `OTOptionManagement` function to request the option values specified in the string. Finally, it calls the `OTCreateOptions` function to create the options buffer. The `OTCreateOptions` function creates the `TOption` structures and places them in the buffer, making sure that the structures are properly aligned.

Listing 5-2 Constructing an options buffer using the `OTCreateOptions` function

```
char* myStr = "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", &myStr, &cmd.opt)
```

In this case, the initial value of `cmd.opt.len`, which is 0, tells the `OTCreateOptions` function at what offset it should begin to append option information in the buffer. When the function returns, this field specifies the actual length of the buffer.

Setting Default Values

To set all of an endpoint's options to their default values, call the `OTOptionManagement` function, specifying `T_NEGOTIATE` for the `flags` field and

allocating a buffer containing only one option named T_ALLOPT. Doing this saves you the trouble of constructing a TOption structure for every option the endpoint supports. However, there is no guarantee that the provider can honor your request simply because you request default values. Therefore, you must allocate a buffer that is large enough to hold the option values returned in the ret parameter.

Allowing the Endpoint Provider to Select an Option Value

You can specify that an endpoint provider selects an appropriate option value by setting the endpoint's value field to the constant T_UNSPEC. This is especially useful in complex options such as ISO throughput where the option value has an internal structure.

Retrieving Option Values

This section describes how you can retrieve information about options, including obtaining current and default option values for an endpoint and obtaining current option values related to a connection, transaction, or datagram.

When retrieving option values, you must allocate a buffer that is large enough to contain the options when the function returns. The section "Obtaining the Maximum Size of an Options Buffer" on page 5-18 explains how you do this.

Obtaining Current and Default Values

To obtain some of an endpoint's default or current option values, you call the OTOptionManagement function. You specify T_DEFAULT or T_CURRENT for the flags field of the req parameter, and you use the option.buf field to specify the option names in which you are interested. When the function returns, it places TOption structures, describing the default or current option values, in the buffer referenced by the opt.buf field of the ret parameter.

If you are interested in obtaining all of an endpoint's default or current values, you can use the following methods:

- To obtain an endpoint's default values, call the OTOptionManagement function, specifying T_DEFAULT for the flags field and T_ALLOPT for the option name.

- To obtain an endpoint's current option values, call the `OTOptionManagement` function, specifying `T_CURRENT` for the `flags` field and `T_ALLOPT` for the option name.

Using `T_ALLOPT` for the option name allows you to construct an input buffer that contains only one option. Remember, however, that you must allocate an output buffer that is large enough to hold all of an endpoint's option values when the function returns.

Retrieving Values for Connection-Oriented Endpoints

When you are establishing a connection, it is possible to negotiate association-related option values at every point in the connection process, as illustrated in Figure 5-1 on page 5-6. Both the active and passive peers might want to retrieve option values during this process.

The passive peer might want to know the proposed option values under negotiation. It can retrieve these by calling the `OTListen` function. After examining the option values returned by the `OTListen` function, the passive peer can negotiate option values by specifying the desired option values with the `OTAccept` call used to accept the connection. Using this method, the passive peer can examine the requested option values before proposing alternate values.

The passive peer can also negotiate alternate values by using the `OTOptionManagement` function to preset option values for the endpoint accepting the connection. This sets the current option values for the endpoint so that when the passive peer calls the `OTAccept` function, these are the option values that are negotiated with the requested values.

The passive peer can try to negotiate option values that are higher than the proposed values. The outcome depends on the protocol. If the protocol rejects the new option values, the connection fails, and the endpoint provider issues a `T_DISCONNECT` event. Depending on timing and the implementation, the `OTAccept` function either succeeds or fails with the `kOTLookErr` result.

The association-related options retrieved by the passive peer are related to the incoming connection, identified by a sequence number, and are not related to the listening endpoint. Option values currently effective for the listening endpoint might affect the values retrieved by the `OTListen` function because the endpoint is involved in the negotiation process, but these values are not the same as the option values related to the connection request. That is to say, calling the `OTOptionManagement` function to retrieve the option values that were

currently effective for the listening endpoint is likely to yield a different set of values than you would find by examining the values of options passed in the `call` parameter to the `OTListen` function.

When you establish the connection—that is, when a synchronous call to the `OTConnect` function returns or when the active peer calls the `OTRcvConnect` function— all final negotiated values effective for the connection are returned in the buffer passed in the `rcvCall` or `call` parameter, respectively. These option values include 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. Options specified on input to the `OTConnect` call that are not supported or that refer to an unknown protocol are ignored and not returned by the `OTConnect` or `OTRcvConnect` function when it returns.

Retrieving Values for Connectionless Transactionless Endpoints

You can retrieve association-related options set for connectionless transactionless endpoints by examining the buffer passed in the `udata` parameter to the `OTRcvUData` function. These options relate to the incoming datagram, not to the endpoint receiving it. For example, the IEEE 802.2 protocol uses option values to specify whether a datagram is a multicast or broadcast packet.

Because the options you retrieve are related to the datagram and not to the listening endpoint, their number and values can change with every transmission.

Because you are receiving information—that is, you are simply reading the contents of the options buffer—you can ignore the `status` field for these options.

Retrieving Values for Connectionless Transaction-Based Endpoints

You can retrieve association-related options set for connectionless transaction-based endpoints by examining the buffer passed in the `req` parameter to the `OTRcvURequest` function. These options relate to the current transaction, not to the endpoint receiving the request. Consequently, options and their values can change with each transaction.

Because you are receiving information—that is, you are simply reading the contents of the options buffer—you can ignore the `status` field for these options.

Parsing an Options Buffer

If you use the `OTOptionManagement` function to set, verify, or retrieve values, the function returns in the `ret` parameter a pointer to a buffer containing option information. You can use the `OTCreateOptionString` function to parse this buffer and create a string that lists all options and their current values.

The code fragment shown in Listing 5-3 calls the `OTOptionManagement` function to retrieve the option values currently effective for an endpoint. On return, the `OTOptionManagement` function stores these in the `cmd` structure. Next, the code calls the `OTCreateOptionString` function. The first input parameter, `"SerialA"`, specifies the name of the protocol. The next input parameter, `opts`, is a pointer to the buffer containing the option values returned by the `OTOptionManagement` function. The expression `cmd.opt.buf + cmd.opt.len`, which provides the next input parameter, specifies the length of the buffer. Using this information, the `OTCreateOptionString` function returns a string containing each option name and its respective value. The final parameter to the `OTCreateOptionString` function specifies the length of the string.

Listing 5-3 Using the `OTCreateOptionString` function to parse through a buffer

```
TOptMgmt        cmd;
UINt8           myBuffer[512];
char            myString[256]

cmd.opt.len = sizeof(TOption);
cmd.opt.maxlen = sizeof(myBuffer);
cmd.opt.buf = myBuffer;
((TOption*) buffer)->len = sizeof(TOption);
((TOption*) buffer)->level = COM_SERIAL;
((TOption*) buffer)->name = T_ALLOPT;
((TOption*) buffer)->status = 0;
cmd.flags = T_CURRENT;

OTOptionManagement(theEndpt, &cmd, &cmd);

TOption* opts = (TOption*)cmd.opt.buf;
err = OTCreateOptionString("SerialA", &opts,
        cmd.opt.buf + cmd.opt.len, string, sizeof(string));
printf("Options = \"%s\"", string);
```

Note
The `OTCreateOptionString` function is supplied solely as a debugging aid. You should not include the function in a production version of your application because there is no provision made for localizing string information. ◆

Verifying Option Values

In addition to obtaining default or current values and negotiating new values, you can use the `OTOptionManagement` function to verify whether an endpoint supports one or more options. To do this, you construct a buffer containing `TOption` structures describing the options you are interested in and pass this buffer in the `req` parameter to the `OTOptionManagement` function, specifying `T_CHECK` for the action flag. When the function returns, you can examine the `status` field of the `TOption` structures for the options passed back to you in the `ret` parameter to determine whether the specified options are supported.

Option Management Reference

This section describes the data types and functions that you use to manage options for endpoint providers and to manipulate option information.

Constants and Data Types

This section describes constants and data types that you use to set and verify options.

XTI-Level Options

Open Transport defines XTI-level options. These options are not association-related; they are negotiated between the client and its endpoint provider. If the protocol you are using supports these options, you can negotiate them while the endpoint is in any state. The protocol level for all of these options is `XTI_GENERIC`. The constant names used to specify XTI-level options are given by the following enumeration:

```
enum
    {
        XTI_DEBUG           = (OTXTIName)0x0001,
        XTI_LINGER          = (OTXTIName)0x0080,
        XTI_RCVBUF          = (OTXTIName)0x1002,
        XTI_RCVLOWAT        = (OTXTIName)0x1004,
        XTI_SNDBUF          = (OTXTIName)0x1001,
        XTI_SNDLOWAT        = (OTXTIName)0x1003,
        XTI_PROTOTYPE       = (OTXTIName)0x1005
    };
```

Constant Descriptions

`XTI_DEBUG` A constant specifying whether debugging is enabled. Debugging is disabled if the option is specified with no value. This option is an absolute requirement.

`XTI_LINGER` A value defined by a linger structure (page 5-31) that specifies whether the option is turned on (`T_YES`) or off (`T_NO`) and specifies a linger period in seconds. This option is an absolute requirement.

You use this option to extend the execution of the `OTCloseProvider` function for some specified amount of time. The delay allows data still queued in the endpoint's internal send buffer to be sent before the endpoint provider is closed. If you call the `OTCloseProvider` function and the send buffer is not empty, the endpoint provider attempts to send the remaining data during the linger period, before closing. Open Transport discards any data remaining in the send buffer after the linger period has elapsed.

Consult the documentation for your protocol to determine the valid range of values for the linger period.

`XTI_RCVBUF` A positive number specifying the size of the endpoint's internal buffer allocated for receiving data. You can increase the size of this buffer for high-volume connections or decrease the buffer to limit the possible backlog of incoming data.

This option is not an absolute requirement. Consult the documentation for your protocol to determine the valid range of values for the buffer size.

XTI_RCVLOWAT	A positive number specifying the low-water mark for the receive buffer— that is, the minimum number of bytes that must accumulate in the endpoint's internal receive buffer before you are advised that data has arrived via a T_DATA event. Choosing a value that is too low might result in your application's getting an excessive number of T_DATA events and doing unnecessary reads. Choosing a value that is too high might result in buffer overflow and loss of data. This option is not an absolute requirement. Consult the documentation for your protocol to determine the valid range of values for the low-water mark.
XTI_SNDBUF	A positive number specifying the size of the endpoint's internal buffer allocated for sending data. Specifying a value that is too low might result in Open Transport doing more sends than necessary and wasting processor time; specifying a value that is too high might cause flow control problems. This option is not an absolute requirement. Consult the documentation for your protocol to determine the valid range of values for the buffer size.
XTI_SNDLOWAT	A positive number specifying the low-water mark for the send buffer— that is, the minimum number of bytes that must accumulate in the endpoint's internal send buffer before Open Transport actually sends the data. Choosing a value that is too low might result in Open Transport's doing too many sends and wasting processor time. Choosing a value that is too high might result in flow control problems. A value that is slightly lower than the largest packet size defined for the endpoint is a good choice. This option is not an absolute requirement. Consult the documentation for your protocol to determine the valid range of values for the low-water mark.
XTI_PROTOTYPE	The number of the protocol to be used by a RawIP endpoint. For additional information, see the chapter "TCP/IP Services" in this book.

Generic Options

Open Transport defines generic options, and you can use them with any protocol that understands them. The protocol level for each of these options is the same as the name of the protocol that supports them. The constant names used to specify generic options are given by the following enumeration:

```
enum
{
    OPT_CHECKSUM            = (OTXTIName)0x0600,
    OPT_RETRYCNT            = (OTXTIName)0x0601,
    OPT_INTERVAL            = (OTXTIName)0x0602,
    OPT_ENABLEEOM           = (OTXTIName)0x0603,
    OPT_SELFSEND            = (OTXTIName)0x0604,
    OPT_SERVERSTATUS        = (OTXTIName)0x0605,
    OPT_KEEPALIVE           = (OTXTIName)0x0008
};
```

Constant descriptions

`OPT_CHECKSUM` A constant specifying whether checksums are performed. Specify 1 to turn the option on and 0 to turn it off. If you turn it on, a checksum is calculated when a packet is sent and recalculated when the packet is received. If the checksum values match, the client receiving the packet can be fairly certain that data has not been corrupted or lost during transmission. If the checksum values don't match, the function used to receive the packet returns an error.

This option is usually implemented by the lowest-level protocol, although you might be allowed to set it at a higher level. For example, if you use an ATP endpoint, you can set checksumming at the ATP level, even though it is implemented by the underlying DDP protocol.

This option is both association-related and not association-related.

`OPT_RETRYCNT` A positive integer specifying the number of times a function can attempt packet delivery before returning with an error. This option is usually implemented by connection-oriented endpoints or connectionless transaction-based endpoints to enable reliable delivery of data. Such protocols normally set a default value for this option.

	This option is both association-related and not association-related.
OPT_INTERVAL	A positive integer specifying the interval of time that should elapse between attempts to deliver a packet. The number of attempts is defined by the OPT_RETRYCNT option. This option is both association-related and not association-related.
OPT_ENABLEEOM	A constant specifying end-of-message capability. If you set this option, you enable the use of the T_MORE flag with the OTSnd function to mark the end of a logical unit. This option has meaning only for connection-oriented protocols. This option is not association-related.
OPT_SELFSEND	A constant allowing you to send broadcast packets to yourself.
OPT_SERVERSTATUS	A string that sets the server's status. The string's length must be between 0–255 bytes. The maximum length is protocol dependent. This option is used to set the status string that the server returns in response to a client's SendStatus call and is remembered internally on a per-socket basis.
OPT_KEEPALIVE	A keepalive structure (page 5-32) that specifies whether the option is turned on (T_YES) or off (T_NO) and specifies the timeout period in minutes. Connection-oriented protocols can use this option to check that the connection is maintained. If a connection is established but there is no data being transferred, you can specify a time limit within which Open Transport checks to see that the remote end of the connection is still alive. If it is not, Open Transport tears down the connection. This option is association-related.

Status Codes

Open Transport uses status codes to return information about the success of an option negotiation. For individual options, Open Transport returns a status code in the status field of the TOption structure (page 5-33) describing the option. For groups of options negotiated by a single call to the OTOptionManagement function, the function also returns a status code that

specifies the single worst result of the negotiation in the `flags` field of the `ret` parameter.

The constant names that are used to specify information about the outcome of option negotiation are given by the following enumeration:

```
enum
    {
        T_SUCCESS           = 0x020,
        T_FAILURE           = 0x040,
        T_PARTSUCCESS       = 0x100,
        T_READONLY          = 0x200,
        T_NOTSUPPORT        = 0x400
    };
```

Constant descriptions

`T_SUCCESS`	The requested value was negotiated.
`T_FAILURE`	The negotiation failed.
`T_PARTSUCCESS`	A lower requested value was negotiated.
`T_READONLY`	The option was read-only.
`T_NOTSUPPORT`	The endpoint does not support the requested value.

In addition to the status codes given by the status codes enumeration, an option can also have the value `T_UNSPEC` in the `status` field. This indicates that the option does not have a fully specified value at this time. An endpoint provider might return this status code if it cannot currently access the option value. This might happen if the endpoint is in the state `T_UNBND` in systems where the protocol stack resides on a separate host.

Action Flags

The `req` parameter to the `OTOptionManagement` function contains a `flags` field that you set to specify what action the function should take. The constant names that you can specify for this field are given by the following enumeration:

```
enum
    {
        T_NEGOTIATE     = 0x004,
        T_CHECK         = 0x008,
```

```
        T_DEFAULT       = 0x010,
        T_CURRENT       = 0x080
    };
```

Constant descriptions

T_NEGOTIATE	Negotiate the option values specified in the opt.buf field of the req parameter. The overall result of the negotiation is specified by the flags field of the ret parameter. A buffer containing specific negotiated values for each option is referenced by the opt.buf field of the ret parameter.
T_CHECK	Verify whether the endpoint supports the options referenced by the opt.buf field of the req parameter. The overall result of the verification is specified by the flags field of the ret parameter. Specific verification results are returned in the opt.buf field of the ret parameter.
T_DEFAULT	Retrieve the default value for those options in the buffer referenced by the req->opt.buf field. To retrieve default values for all the options supported by an endpoint, include just the option T_ALLOPT in the options buffer. Option values are returned in the opt.buf field of the ret parameter.
T_CURRENT	Retrieve the current value for those options that the endpoint supports and that are specified in the buffer referenced by the req->opt.buf field. To retrieve current values for all the options that an endpoint supports, include just the option T_ALLOPT in the options buffer. Option values are returned in the opt.buf field of the ret parameter.

The Linger Structure

The linger structure specifies the value of the XTI_LINGER option, described in “XTI-Level Options” (page 5-25).

The linger structure is defined by the t_linger data type.

```
struct t_linger
    {
    long l_onoff;       /* option on/off */
    long l_linger;      /* linger time */
    };
```

Field descriptions

`l_onoff` A constant specifying whether the option is turned on (`T_ON`) or off (`T_OFF`).

`l_linger` An integer specifying the linger time, the amount of time in seconds that Open Transport should wait to allow data in an endpoint's internal buffer to be sent before the `OTCloseProvider` function closes the endpoint.

To request the default value for this option, set the `l_linger` field to `T_UNSPEC`.

The Keepalive Structure

The keepalive structure specifies the value of the `OPT_KEEPALIVE` option, described in "Generic Options" (page 5-28).

The keepalive structure is defined by the `t_kpalive` data type.

```
struct t_kpalive
    {
        long kp_onoff;          /* option on/off */
        long kp_timeout;        /* timeout in minutes */
    };
```

Field descriptions

`kp_onoff` A constant specifying whether the option is turned on (`T_ON`) or off (`T_OFF`).

`kp_timeout` A positive integer specifying for how many minutes Open Transport can maintain a connection in the absence of traffic.

The TOption Structure

The `TOption` structure stores information about a single option in a buffer. All functions that you use to change or verify option values use a buffer containing `TOption` structures to store option information. For each option in the buffer, the `TOption` structure specifies the total length occupied by the option information, the protocol level of the option, the option name, the success or failure of a negotiated value, and the value of the option.

You use the `TOption` structure with the `OPT_NEXTHDR` macro, the `OTCreateOptionString` function, the `OTNextOption` function, and the `OTFindOption` function.

The `TOption` structure is defined by the `TOption` data type.

```
struct TOption
    {
        UInt32          len;        /* total length of option        */
        OTXTILevel      level;      /* protocol affected     */
        OTXTIName       name;       /* option name   */
        UInt32          status;     /* status value      */
        UInt32          value[1];   /* data goes here    */
    };
```

Field descriptions

`len`	The size (in bytes) of the option information.
`level`	The protocol for which the option is defined.
`name`	The name of the option.
`status`	A status code specifying whether the negotiation has succeeded or failed. Possible values are given by the status codes enumeration, (page 5-29).
`value`	The option value. To have the endpoint select an appropriate value, you can specify the constant `T_UNSPEC`.

The Option Management Structure

The option management structure is used for the `req` and `ret` parameters of the `OTOptionManagement` function. The `req` parameter is used to verify or negotiate option values. The `ret` parameter returns information about an endpoint's default, current, or negotiated values.

The option management structure is defined by the `TOptMgmt` data type.

```
struct TOptMgmt
    {   TNetbuf     opt;
        OTFlags     flags;
    };
```

Field descriptions

`opt` A `TNetbuf` structure describing the buffer containing option information. The `opt.maxlen` field specifies the maximum size of the buffer. The `opt.len` field specifies the actual size of the buffer, and the `opt.buf` field contains the address of the buffer.

On input—as part of the `req` parameter, the buffer contains `TOption` structures describing the options to be negotiated or verified or contains the names of options whose default or current values you are interested in. You must allocate this buffer, place in it the structures describing the options of interest, and set the `opt.len` field to the size of the buffer.

On output—as part of the `ret` parameter, the buffer contains the actual values of the options you described in the `req` parameter. You must allocate a buffer to hold the option information when the function returns and set the `opt.maxlen` field to the maximum length of this buffer. When the function returns, the `opt.len` field is set to the actual length of the buffer.

`flags` For the `req` parameter, the `flags` field indicates the action to be taken as defined by the action flags enumeration (page 5-30). For the `ret` parameter, the `flags` field indicates the overall success or failure of the operation performed by the `OTOptionManagement` function, as defined by the status codes enumeration (page 5-29).

Functions

This section describes the functions that you can use to determine an endpoint's current and default options or to change them. This section also describes utility functions that you use to manipulate the format of option

information and utility functions that you use to find option information in a buffer.

Determining and Changing Function Values

This section describes the `OTOptionManagement` function, which you use to obtain information about an endpoint's default or current option values and to change these values if needed.

OTOptionManagement

Determines an endpoint's current or default option values or changes these values.

C INTERFACE

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

C++ INTERFACES

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

PARAMETERS

`ref`	The endpoint reference of the endpoint for which you are checking or setting option values.
`req`	A pointer to an option management structure (page 5-33), which describes the action to be taken by the function and the options affected.
`ret`	A pointer to an option management structure (page 5-33), which describes the options that were changed or returned by the function and how successful the negotiation process was.

DESCRIPTION

To use the `OTOptionManagement` function, you must have opened an endpoint using the `OTOpenEndpoint` or `OTAsyncOpenEndpoint` functions.

You use the `OTOptionManagement` function to negotiate, retrieve, or verify an endpoint's protocol options. If the endpoint is in asynchronous mode and you have not installed a notifier function, it is not possible to determine when the function completes.

The action taken by the `OTOptionManagement` function is determined by the setting of the `req->flags` field. The following bulleted items describe the different operations that you can perform and the flag settings that you use to specify these operations.

- To negotiate values for the endpoint, you must call the `OTOptionManagement` function, specifying `T_NEGOTIATE` for the `req->flags` field. The endpoint provider evaluates the requested options, negotiates the values, and returns the resulting values in the option management structure pointed to by the `ret->opt.buf` field. The `status` field of each returned option is set to a constant that indicates the result of the negotiation. These constants are described by the status codes enumeration (page 5-29).

  For any protocol specified, you can negotiate for the default values of all options supported by the endpoint by specifying the value `T_ALLOPT` for the `name` field of the `TOption` structure. This might be useful if you want to change current settings or if negotiations for other values have failed. The success of the negotiations depends partly on the state of the endpoint—that is, simply because these are default values does not guarantee a completely successful negotiation. When the function returns, the resulting values are returned, option by option, in the buffer pointed to by the `ret->opt.buf` field.

- To retrieve an endpoint's default option values, call the `OTOptionManagement` function, specifying `T_DEFAULT` for the `req->flags` field. You must also specify the name of the option (but not its value) in the `TOption` structure that you create for each of the options you are interested in.

  When the function returns, it passes the default values for the options back to you in the buffer pointed to by the `ret->opt.buf` field. For each option, the `status` field contains `T_NOTSUPPORT` if the protocol does not support the 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 the `ret->flags` field. The meaning of this result is described by the status codes enumeration (page 5-29).

When getting an endpoint's default option values, you can specify `T_ALLOPT` for the option name. This returns all supported options for the specified level with their default values. In this case, you must set the `opt.maxlen` field to the maximum size required to hold an endpoint's option information. The `info.opt` field of the `TEndpointInfo` structure specifies the maximum size of a buffer used to hold option information for an endpoint.

- To retrieve an endpoint's current option values, call the `OTOptionManagement` function, specifying `T_CURRENT` for the `req->flags` field. For each option in the buffer referenced by the `req->opt.buf` field, specify the name of the option you are interested in. The function ignores any option values you specify.

  When the function returns, it passes the current values for the options back to you in the buffer referenced by the `ret->opt.buf` field. For each option, the `status` field contains `T_NOTSUPPORT` if the protocol does not support the 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 the `ret->flags` field. The meaning of this result is described by the status codes enumeration (page 5-29).

  When retrieving an endpoint's current option values, you can specify `T_ALLOPT` for the option name. The function returns all supported options for the specified protocol, with their current values. In this case, you must set the `opt.maxlen` field to the maximum size required to hold an endpoint's option information. The `info.opt` field of the `TEndpointInfo` structure specifies the maximum size of a buffer used to hold option information for an endpoint.

- To check whether an endpoint provider supports certain options or option values, you must call the `OTOptionManagement` function, specifying `T_CHECK` for the `req->flags` field. Checking options or their values does not change the current settings of an endpoint's options.

  - To check whether an option is supported, set the `name` field of the `TOption` structure to the option name, but do not specify an option value. When the function returns, the `status` field for the corresponding `TOption` structure in the buffer pointed to by the `ret->opt.buf` field is set to `T_SUCCESS` if the option is supported, `T_NOTSUPPORT` if it is not supported or needs additional client privileges, and `T_READONLY` if it is read-only.
  - To check whether an option value is supported, set the `name` field of the `TOption` structure to the option name, and set the `value` field to the value you want to check. When the function returns, the `status` field for the

corresponding TOption structure in the buffer pointed to by the ret->opt.buf field is set as it would be if you had specified the T_NEGOTIATE flag. The overall result of the option checks is returned in the ret->flags field, which contains the single worst result of the option checks. The meaning of this result is described by the status codes enumeration (page 5-29).

SPECIAL CONSIDERATIONS

While an option management call is outstanding, any other functions that are called for the same endpoint return with a kOTStateChangeErr result.

If the endpoint is in asynchronous mode, the provider might issue the T_OPTIONMGMTCOMPLETE event before the function returns the first time.

COMPLETION EVENTS

T_OPTMGMTCOMPLETE	0x20000006	The OTOptionManagement function has completed. The cookie parameter of the notifier function points to the ret parameter.

SEE ALSO

Option information is formatted using the TOption structure (page 5-33). For additional information about the format of the options buffers, see "Specifying Option Values" (page 5-18).

For more information about the OTOpenEndpoint and OTAsyncOpenEndpoint, see the reference section of the chapter "Endpoints" in this book.

For additional information about using the T_ALLOPT option, see "Setting Default Values" (page 5-20) and "Obtaining Current and Default Values" (page 5-21).

For more information about creating the buffer referenced by the req->opt.buf field, see the description of the OTCreateOptions function, next.

For information about creating a string referenced by the ret->opt.buf field, see the description of the OTCreateOptionString function (page 5-42).

Manipulating the Format of Option Information

You use the Open Transport utility functions described in this section to construct a buffer describing option values from a string or to create a string from a buffer containing option values. You do not have to create an endpoint to use Open Transport utility functions, but you do have to initialize Open Transport as described in the chapter "Configuration Management" in this book.

OTCreateOptions

Writes option information into a buffer, from a string specifying option values.

C INTERFACE

```
OSErr OTCreateOptions (const char* prtclName; char** strPtr,
                       TNetbuf* buf);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`prtclName` The name of the protocol for which the option is set. For example, for an AppleTalk endpoint this might be "atp" or "ddp."

`strPtr` A pointer to a pointer to a string containing option information. If an error occurs in writing the option information to the buffer, `strPtr` points to the position in the string where the error occurred.

Open Transport maintains an internal database relating to options and their values. Open Transport might not be able to write option information to the buffer because it cannot match a name or value you have specified with a name or value in its database. This is either because you misspelled a name or

specified a value that is out of range or because the option you want to configure is not included in Open Transport's data base. The latter might be the case for an option that is rarely used.

`buf` A pointer to a `TNetbuf` structure that specifies the size and location of the buffer into which the function writes option information. You must allocate the buffer and set the `buf->opt` field to point to it.

You must set the `buf->maxlen` field to the value specified by the `TEndpoint.options` field for this endpoint. You set `buf->len` to 0. When the function returns, it sets the `buf->len` field to the actual length of the option information, including padding.

The function appends option information to the buffer beginning at the offset specified by the `buf->len` field. Set this field to 0 to start at the beginning of the buffer. When the function returns, the value of the `buf->len` field is updated to reflect the new length.

DESCRIPTION

The `OTCreateOptions` function automates the construction of a buffer that describes endpoint option values for a particular protocol. Given a string, a pointer to a buffer, and the protocol for which the options are set, the function constructs `TOption` structures describing each option specified and then places these structures in the buffer referenced by the `buf->opt` field. After using the `OTCreateOptions` function to construct the buffer, you have most of the information needed to create the `req` parameter to the `OTOptionManagement` function.

The string containing option values has the format:

optionName1 = value optionName2 = value optionName3 = value `[....]`

where *value* can be a numeric value, a string value, or a byte array value. The table below describes how each value is represented.

Format of values	Contents
Numeric	A minus sign (–) prefix for negative numbers, followed by the digits comprising the number; for example, –6784. A $ or 0x prefix for hexadecimal numbers, followed by the digits comprising the number; for example, $FFFE.
String	The option string, which is composed of a delimiter character, followed by the characters comprising the string, followed by the delimiter character. A delimiter character is the first non blank character after the equals sign. For example, SomeOptionName = *The String Option*, or SomeOtherOptionName = %Another String Option%.
Byte array	A leading $ or 0x followed by a sequence of hex digits with no intervening spaces or tabs. There must be an even number of digits; for example, $FF12EE46.

Possible values for option names are given in the documentation for the protocol you are using. Generic option names are described in "XTI-Level Options and General Options" on page 5-10.

SEE ALSO

You use the buffer constructed by the `OTCreateOptions` function as part of the `req` parameter to the `OTOptionManagement` function (page 5-35). Listing 5-2 on page 5-20 shows how you use the `OTCreateOptions` function.

You use the `OTCreateOptionString` function, described in the next section, to reverse the process and construct a string containing endpoint option values by parsing a buffer containing `TOption` structures.

OTCreateOptionString

Creates a string from a buffer containing `TOption` structures.

C INTERFACE

```
OTCreateOptionString (const char* prtclName, TOption** optPtr,
                      void* bufEnd, char* string,
                      size_t stringSize);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`prtclName`	A constant specifying the name of the protocol for this option or options.
`optPtr`	A pointer to a pointer to a buffer containing one or more `TOption` structures.
`bufEnd`	A pointer to the first byte of memory past the last option.
`string`	A pointer to a buffer where the string is to be stored. You must allocate this buffer.
`stringSize`	The length of the buffer where the string is to be stored. You must specify this value.

DESCRIPTION

You can use the `OTCreateOptionString` function to parse through the options buffer returned by the `ret` parameter to the `OTOptionMangement` function and create a string specifying option values that you can display.

This function is supplied solely as a debugging aid. You should not include the function in a production version of your application because there is no provision made for localizing string information.

SEE ALSO

You obtain the buffer to be converted from the `ret` parameter to the `OTOptionManagement` function (page 5-35). Listing 5-3 on page 5-24 shows how you use the `OTCreateOptionString` function.

You can reverse the procedure and build an options buffer from a string by using the `OTCreateOptions` function (page 5-39).

Finding Options

You use the two functions described in this section to find a specific option in an options buffer or to find the next option in the buffer. You do not have to create an endpoint to use these functions, but you do have to initialize Open Transport as described in the chapter "Configuration Management" in this book.

OTFindOption

Finds a specific option in an options buffer.

C INTERFACE

```
TOption* OTFindOption (UInt8* buffer, UInt32 buflen,
                       OTXTILevel level, OTXTIName name);
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`buffer`	A pointer to the buffer containing the option to be found.
`buflen`	The size of the buffer containing the option to be found.
`level`	The protocol of the option to be found.
`name`	The name of the option to be found.

DESCRIPTION

Given a buffer such as might be returned by the `OTOptionManagement` function or by any endpoint function that returns a buffer containing option information, you can use the `OTFindOption` function to find a specific option in the buffer.

SEE ALSO

To parse through a buffer, option by option, use the `OTNextOption` function (described next).

To convert option information in a buffer into a string, use the `OTCreateOptionString` function (page 5-42).

OTNextOption

Locates the next `TOption` structure in a buffer.

C INTERFACE

```
OSErr OTNextOption (UInt8* buffer, UInt32 buflen,
                    TOption** prevOptPtr;
```

C++ INTERFACE

None. C++ applications use the C interface to this function.

PARAMETERS

`buffer`	A pointer to the buffer containing the option to be found.
`buflen`	A long specifying the size of the buffer containing the option to be found.
`prevOptPtr`	A pointer to a pointer to the first or current `TOption` structure. The first time you call the function, set this parameter to the beginning address of the buffer containing the options to be found. On return, this parameter references the beginning address of the next option

DESCRIPTION

The `OTNextOption` function allows you to parse through a buffer containing `TOption` structures describing an endpoint's option values. Within the buffer, `TOption` structures are aligned to long-word boundaries. This function takes into account this padding when it calculates the beginning address of the next `TOption` structure and it returns that address in the `prevOptPtr` parameter.

The first time you call the option, set the `prevOptPtr` parameter to the beginning address of the buffer. When the function returns, the `prevOptPtr` parameter points to the next (second) option in the buffer. You can continue this process, specifying the value returned for the `prevOptPtr` parameter by the previous invocation of the function, each time you call the function to obtain the beginning address of each option in the buffer.

SEE ALSO

To find a particular option in a buffer, use the `OTFindOption` function (page 5-43).

Configuration Management

Contents

This chapter describes Open Transport functions that initialize Open Transport, configure a provider, and provide information about the ports available on your computer.

You need to read this chapter if your application wants to use Open Transport or open a provider because in the former case, you must initialize all or some of Open Transport, and in the latter case, you must create a provider configuration. In addition, if your application has the ability to switch ports, you need to be able to obtain port information.

This chapter discusses

- initializing all or part of Open Transport
- configuring providers and reusing configuration structures
- browsing available ports and getting specific port information
- registering as an Open Transport client
- handling yield port requests

This chapter begins by introducing the basic concepts of provider configuration and port information, then gives the details of how to initialize Open Transport, how to find a specific port and extract information about it, and how to register your application as an Open Transport client.

About Provider Configurations

Before you can open a provider, you must first tell Open Transport how to configure it with the protocol and options you want the provider to use. To do this, you pass a string to a function (`OTCreateConfiguration`) that creates a configuration structure (of data type `OTConfiguration`) describing the service you want.

The configuration string can be the name of a single protocol, such as “adsp”, “tcp”, or “dnr,” or it can be a full comma-separated list of protocol and port names, with option values specified in parentheses after the name of the protocol to which they apply. For instance,

```
"adsp,ddp,ltlkB"
```

describes an AppleTalk Data Stream Protocol (ADSP) endpoint provider using the Datagram Delivery Protocol (DDP) and with LocalTalk link access provided through the default port (the LocalTalk B printer port).

Open Transport has internally defined defaults for how protocols can be layered upon each other. If you give Open Transport a single protocol name, it checks its defaults to determine which lower layers are missing. Thus, the shorter string

```
"adsp"
```

also describes an identical ADSP endpoint provider. Likewise, if you skip a protocol layer in the string, Open Transport uses its defaults to try to complete it. For instance, the specification “tcp, enet” is incomplete because the Transmission Control Protocol (TCP) does not have direct access to Ethernet, and so Open Transport puts the default Internet Protocol (IP) between TCP and Ethernet.

You can also specify an option as part of the configuration string. To do this, you need to know which protocols use which options and how to translate the option’s constant name, given in the header files, into a string that the configuration functions can parse. See the TCP/IP and AppleTalk chapters for lists of their procotol-specific options and their equivalent string values, but for a simple example,

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

describes an ADSP endpoint provider with the DDP checksum option enabled.

If you want to identify a particular port in the configuration string, you use the port name to do so (described in the next section). More typically, however, you leave this value blank—for example, using only “adsp” or “adsp, ddp,” which configures the provider with whatever port is specified in the associated control panel.

Most protocols have a hardcoded string value that you can use to configure providers. For example, DDP uses “ddp” and ADSP uses “adsp.” There are also constants that identify each protocol, such as `kDDPName` and `kADSPName`. For a complete list of the AppleTalk constant-string equivalents, see the chapter “Introduction to AppleTalk” in this book. For a TCP/IP service provider, you can use the constant `kDefaultInternetServicesPath`; there is no hardcoded equivalent.

You can use the constant or the hardcoded value to create providers that do not use options and that adhere to the default procotol layering. For example, to configure a fairly straightforward DDP endpoint, you could use either of the following lines of code:

```
OTOpenEndpoint(OTCreateConfiguration("ddp"), 0, NULL, &err)

OTOpenEndpoint(OTCreateConfiguration(kDDPName), 0, NULL, &err);
```

To configure more complex providers, it is easier to use the hardcoded strings. Using the constant can be confusing, as compared in the following lines of code:

```
OTOpenEndpoint(OTCreateConfiguration
        ("adsp(EnableEOM=1),ddp,ltlkB"), 0, NULL, &err)

OTOpenEndpoint(OTCreateConfiguration
        (kADSPName"(EnableEOM=1),"kDDPName",ltlkB"), 0, NULL, &err);
```

Note
The `OTCreateConfiguration` function returns a pointer to the configuration structure it creates. You pass this pointer as a parameter to the open-provider functions such as the `OTOpenEndpoint` or `OTOpenMapper` functions (discussed in the chapters "Endpoints" and Mappers" in this book). ◆

About Port Information

Central to Open Transport's architecture is the concept of a port. In Open Transport, a **port** is a logical entity that combines a hardware device and the software driver that acts as an interface to it. Ethernet, serial devices, and LocalTalk ports are examples of ports commonly used in Open Transport.

Typically, your application uses whichever port is defined in the AppleTalk or TCP/IP control panel. If, however, your application provides special port manipulation features, you need the additional port information data structures, constants, and functions that Open Transport provides for browsing among the ports available to your computer and for finding specific ports.

Open Transport provides a standard naming scheme for describing the ports available to a computer. There are three ways to identify each port uniquely: its port name, its module name, and its port reference.

The **port name** is a unique name that designates the port. This name identifies the port without using any location information. For instance, `"ltlkA"` identifies LocalTalk on the serial port, and `"ltlkB"` identifies LocalTalk on the modem port. This name must always be used in the path string for `OTCreateConfiguration` to uniquely identify a port.

The port name is typically an abbreviation of the port's device type plus a suffix, usually numeric, such as "enet0," "enet1," and "enet2." For historic reasons, LocalTalk and serial ports use an alphabetic suffix instead. For example, "ltlkA" is the modem port and "ltlkB" is the printer port. The port name is a zero-terminated string that can have a maximum length of 36 bytes: 31 bytes for the name, up to 4 bytes of extra characters (called *minor numbers* in XTI specifications) that are currently not used, and a byte for the terminating zero.

Each port on a computer also has a **module name,** which is the name of the actual Streams module that implements the driver for this port. You don't use this name; Open Transport uses this name internally.

You can also uniquely identify a port with a **port reference,** which is a 32-bit value that describes a port's hardware characteristics: its device and bus type, its physical slot number, and, where applicable, its multiport identifier. For details of the possible values you can use in a port reference, see the section "The Port Reference," beginning on page 6-19. Open Transport allows clients to use a device name to specify a port. In this case, Open Transport uses the first device of that type that is registered and available. For most devices, this means the motherboard device, if one exists; if one doesn't, Open Transport uses the first slotted device that was registered.

The **multiport identifier** is a port function parameter that distinguishes between multiple ports when a single slot supports more than one port. This parameter, called `other`, is part of the port reference structure, which is described in the section "The Port Reference" on page 6-19.

Typically, the hardware device in a multiport slot is either a plug-in multifunction card with multiple ports on it or a device with multiple uses, one or more of which is a port. Examples of multifunction cards are a motherboard with onboard Ethernet and the SerialNB card with its four ports; an example of a multi-use device on most Macintosh computers is the SCC chip that can handle both LocalTalk and serial communications. Typically, a multifunction

card has multiple ports that use different values for the `other` parameter and possibly different device attributes, and a multi-use device is registered with all attributes identical except for the device type.

There's a special type of port, called a **pseudodevice,** that is a driver that doesn't interface to a hardware device; instead, it interfaces to other device drivers. Pseudodevices are provided as a convenience for the Open Transport architecture. Open Transport defines special device types for certain common pseudodevices, such as modem, PPP, and SLIP. Because Open Transport can't possibly accomodate all possible pseudodevices, there is a generic device type, designated with the constant `kOTPseudoDevice`, that identifies unknown or unusual pseudodevices. Each pseudodevice must have a unique port reference. Typically, a pseudodevice is private, and a flag indicating that the port is private notifies applications browsing the **port registry** that the port is not normally available for public use. The port registry is a registry of ports that Open Transport creates when it scans the network for all available ports.

Every port on the computer is described in Open Transport by a **port structure,** which contains its port reference, several sets of information flags, its port name, its Streams module name, and the slot ID (for ports on a PCI bus). For details of the port structure, see the section "The Port Structure," beginning on page 6-17.

The port structure includes fields that allow you to identify a port's **child port,** which allows you to identify which of several available hardware devices the port uses. A port may have more than one child port, all of which can be active simultaneously.

For example, in many implementations, a SLIP port is a pseudodevice that uses a serial port as its hardware device. If more than one serial port is available, the SLIP pseudodevice could use any of them. A SLIP port therefore always has a serial port as its child port so that when multiple serial ports are available, you can use the child port information to find out which serial port the SLIP port is using. Other device types, such as fast ethernet devices, do not have child ports because they have a one-to-one relationship with their hardware device—that is, they have only one possible choice for the hardware device they can use.

The **slot ID** is a user-visible identifier used for cards on PCI bus computers. To derive this value, Open Transport accesses information in the system registry. The **system registry,** sometimes referred to as the *Name Registry,* is a register of hardware and software configuration information for Power Macintosh computers that is maintained by Mac OS. For more information about the

system registry, or Name Registry, see *Designing PCI Cards and Drivers for Power Macintosh Computers.*

One set of flags indicate a port's framing capabilities—that is, the different packet headers and trailers (data frames) permitted by the protocol on that port. The framing flags are specific to the device type being registered. See the appropriate documentation for the device to determine how to interpret them.

For each hardware device type, Open Transport derives a default port name based on the port name by stripping its numeric (or alphabetic, in the case of LocalTalk and serial ports) suffix. All ports on a computer that are the same hardware device type result in the same default port name. Thus, Ethernet devices default to "enet." For all hardware device types, you can use the default port name as part of the configuration string. If you use a default name such as "enet," Open Transport uses whichever port is identified as the default port. If it can't find that port, OpenTransport returns an error message.

In the case of LocalTalk, however, Open Transport uses a flag to define a specific port as a **port alias,** or a *default* port, for LocalTalk ports. This port is called "ltlk" and uses the same Streams module name as the default LocalTalk port. Normally, the LocalTalk default port is the printer port, "ltlkB," but if a computer doesn't have an "ltlkB" port, then the LocalTalk default is the modem port, "ltlkA." Because both the port alias and the default port have the same Streams module name, when you use the port alias to configure the port, Open Transport can locate the default port even if a port doesn't use the standard "ltlkB" default.

Using the Configuration Functions

This section describes how to determine whether Open Transport is available, how to initialize all or some of Open Transport, how to configure providers, how to obtain port information, and how to register as an Open Transport client.

Determining Whether Open Transport Is Available

If you want to know if Open Transport is available on your computer, use the `Gestalt` function with `'otan'` as its selector. If `Gestalt` returns no error and its `response` parameter returns with a value other than 0, Open Transport is

available. To find out whether AppleTalk, TCP, or NetWare are present, you can examine the `response` parameter bits. For a list of the possible bit values, see the section "The Gestalt Selector and Response Bits" on page 6-15.

Initializing Open Transport

There are two Open Transport initialization functions: the `InitOpenTransport` and `InitOpenTransportUtilities` functions. To initialize all of Open Transport, you call the `InitOpenTransport` function, which loads the Open Transport modules.

If your application performs port manipulation and does not need to open or use any providers, you can use the `InitOpenTransportUtilities` function, which initializes only those Open Transport modules that handle ports.

Neither of these two functions, however, loads the AppleTalk or TCP/IP software modules. Open Transport automatically initializes the AppleTalk modules whenever you first open an endpoint, mapper or AppleTalk service provider; and it initializes TCP/IP whenever you open an endpoint or mapper for that protocol family.

Using Open Transport From a Client Application

If your client is an application, you must follow these steps to initialize the Open Transport software:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. If you use the Apple Shared Library Manager (ASLM), call the `InitLibraryManager` function.
3. Call the `InitOpenTransport` (or `InitOpenTransportUtilities`) function.

When you are no longer using Open Transport, you can choose to unload the Open Transport software modules by using the `CloseOpenTransport` function. If you used ASLM, you can then call the `CleanupLibraryManager` function. Both of these functions are optional; the system automatically calls them if your application does not call them.

Using Open Transport From a Stand-Alone Code Segment

If your client is a stand-alone code segment or code fragments, you must follow these steps to initialize the Open Transport software:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. Establish an A5 world if you are running on a 68000-family Macintosh computer. See the *Apple Shared Library Manager Developer's Guide* for details of how to do this.
3. If you use ASLM, call the `InitLibraryManager` function.
4. Call the `InitOpenTransport` (or `InitOpenTransportUtilities`) function.

Note
Stand-alone code segment clients that are on 68000-family Macintosh computers have to ensure that their A5 world is correct each time they call an Open Transport function. ◆

When you are no longer using Open Transport, you can unload the Open Transport software modules. For stand-alone code segments, this means that you must call the `CloseOpenTransport` function before you unload from memory, and if you used ASLM, you must call the `CleanupLibraryManager` function.

System software cannot unload Open Transport until the last software module on your computer that called the `InitOpenTransport` or `InitOpenTransportUtilities` function has also called the `CloseOpenTransport` function.

Creating and Reusing Provider Configurations

Once Open Transport is initialized, you need to configure any providers you want to use for transmitting and receiving data. To do this, you create a configuration structure with the `OTCreateConfiguration` function using a configuration string, as described in the section "About Provider Configurations," beginning on page 6-3.

You typically call the `OTCreateConfiguration` function inline while calling a function that creates and opens a provider (for example, the `OTOpenEndpoint` function). The function that opens providers checks whether the `OTCreateConfiguration` function returned `((OTConfiguration*)-1L)` or `NULL`, and if so, returns an appropriate result code.

The open-provider functions take a pointer to the configuration structure as input, but as part of their processing, they destroy the original configuration structure. Since typically you use the `OTCreateConfiguration` function to create only a single provider at a time, this works fine most of the time. Occasionally,

however, you may want to reuse a configuration structure to create a second identical provider, or you may want to reuse a configuration sent from another application for which you do not have the configuration string.

The only way to reuse a configuration structure is to clone it with the `OTCloneConfiguration` function before opening your first provider. In this way, you can save the provider's configuration to disk or make multiple copies of the same configuration.

For example, you might have only a pointer to a structure, but you want to create ten endpoints, and so you need ten structures. The moment you use the original pointer to create an endpoint, the structure is gone. You can't call the `OTCreateConfiguration` function because you don't have the original configuration string; you were only passed the structure. However, if you can create a copy of the structure, you don't need the string; so you clone the original structure nine times before opening the first endpoint, which results in a total of ten identical configuration structures.

Obtaining Port Information

If your application manipulates ports, you may need port information to locate a specific port or to find out how what ports are registered for your computer. Open Transport registers all ports that are associated with your computer and creates a port structure for each port. You can then use the various Open Transport port functions to access these structures and get information from them. (The port structure is described in the section "The Port Structure," beginning on page 6-17.)

If you want to find out the port associated with a given provider, you can use the `OTGetProviderPortRef` function. If you don't know which port structure you want or if you want to provide a list of user-readable port names to your user, you can use the `OTGetIndexedPort` function to iterate through all the ports available on a computer, obtaining the port structure of each.

There are also two find functions you can use to find the port structure for a specific port: If you know its port name, you can use the `OTFindPort` function, or if you know its port reference, you can use the `OTFindPortByRef` function.

If you want to use the `OTFindPortByRef` function, you need a port reference. There are several ways you can get one: Another application might have passed it to you, another application could have put it into a port structure that you now access by using the `OTGetIndexedPort` function, or you can create one.

To create a port reference, you use the `OTCreatePortRef` function. You must know all the port's hardware characteristics: its device and bus type, its slot number, and its multiport identifier (if it has one). You cannot use wildcards to fill in any element you don't know. Possible device and bus types are described in "The Port Reference," beginning on page 6-19.

Note
Note that the slot numbers for NuBus™ cards are physical; that is, they are the slot numbers returned by the Slot Manager and not the slots seen in various network configuration applications. Physical slot numbers depend on the type of card installed. For example, NuBus cards number their slots 9 to 13, which appear in the AppleTalk or TCP control panels as slots 1 to 5. For PCI cards, however, the slot numbers are their logical slot IDs as defined in the port structure. For cards in a PCI bus, it is not possible, a priori, to create a port reference that corresponds to a known card, so applications must iterate through the port registry to find appropriate PCI ports. ◆

For example, if you want to find out the port name of the Ethernet port in NuBus slot 13, you can use this line of code to create a port reference for this port:

```
OTPortRef ref = OTCreatePortRef(kOTNuBus, kOTEthernetDevice, 13, 0);
```

If you then pass the result of this call to the `OTFindPortByRef` function, `OTFindPortByRef` fills a buffer with the port structure that has this port reference and returns a pointer to the buffer. You can examine the port structure's fields for its port name.

Open Transport has predefined variants of the `OTCreatePortRef` function for the most commonly used hardware devices such as the NuBus, PCI, and PCMCIA devices. These are found in the section describing the function `OTCreatePortRef` (page 6-36).

If you want to extract information from a port reference, you have to use specific Open Transport functions: `OTGetDeviceTypeFromPortRef`, `OTGetBusTypeFromPortRef`, and `OTGetSlotFromPortRef`.

Requesting a Port to Yield Ownership

There may be times when you need to use a particular port that is owned by another provider. You can use the `OTYieldPortRequest` function to request the owner of a port (normally, a serial port or modem) to yield the use of the port to you. Open Transport then issues a `kOTYieldPortRequest` event to each provider of any registered clients for that port for acceptance or refusal. If the owner has not registered as a client of Open Transport, no event can be sent and acceptance is implicit.

If the current owner wants to deny the request, it puts a negative error code into the `fDenyReason` field in the port close structure indicating its reason for refusal. The `OTYieldPortRequest` function then returns with this error code as its result and with a buffer listing all the clients that have refused the request, (normally only one).

If the `OTYieldPortRequest` function returns without an error, the port is available for your use. You can then bind it with a queue length (`qlen`) greater than 0 or establish a connection with it. If you don't use the port within 10 seconds, the port automatically stops being available for your use and reverts to its original owner.

You can force a passive client to yield by using a value of `NULL` in the `OTYieldPortRequest` function's `buffer` parameter. When the function returns without an error, the port is available. Note that a port can only be yielded in this manner if its current client is passively listening; it cannot be yielded if a connection is in progress.

Providers owned by unregistered clients need to be prepared to receive `kOTProviderIsDisconnected` and `kOTProviderIsReconnected` events when the connection between the provider and port is unexpectedly disconnected and reconnected due to a successful yield reques.

Registering as an Open Transport Client

You can use the `OTRegisterAsClient` function to register your application as an Open Transport client and provide Open Transport with a notifier function for sending messages to you. Once you are registered as a client, Open Transport can notify you of system events, such as the port transition events that occur when a particular port is disabled or closed and when it is reenabled. By registering, you also provide Open Transport with a user-readable name to use when informing the user of port transition events.

This function is, however, optional. If you do not want to receive these events, you do not have to call this function.

If you did register your name, when you finish using Open Transport, you need to call the `OTUnregisterAsClient` function to remove your name as an available client. However, the `CloseOpenTransport` function automatically calls this function if you fail to do so.

Configuration Management Reference

This section describes the data types, constants, and functions that you need to initialize Open Transport, configure providers, obtain port information, and register your application as a client.

Constants and Data Types

This section describes the basic configuration management constants, the `Gestalt` function selector and response bits, the configuration structure, the port structure, and the port reference structure.

These constants provide length and size values for modules, provider names, and slot IDs. These fields all end with a byte for the terminating zero. The constant `kMaxProviderNameSize` permits a length of 36 bytes: 31 bytes for the name, up to 4 bytes of extra characters (called *minor numbers* in Streams specifications, and currently not used), and a byte for the zero that terminates the string.

```
enum {
    kMaxModuleNameLength            = 31,
    kMaxModuleNameSize              = kMaxModuleNameLength + 1,
    kMaxProviderNameLength          = kMaxModuleNameLength + 4,
    kMaxProviderNameSize            = kMaxProviderNameLength + 1,
    kMaxSlotIDLength                = 7,
    kMaxSlotIDSize                  = 8,
    kMaxResourceInfoLength          = 31,
    kMaxResourceInfoSize            = 32
};
```

The Gestalt Selector and Response Bits

You can test whether Open Transport and its various parts are available by using the `Gestalt` function with the `'otan'` selector. The `Gestalt` function returns information by setting or clearing bits in the `response` parameter. The bits currently used are defined by constants, shown along with the Open Transport selector in the following enumeration:

```
enum {
    gestaltOpenTpt                          = 'otan',
    gestaltOpenTptPresent                   = 0x00000001,
    gestaltOpenTptLoaded                    = 0x00000002,
    gestaltOpenTptAppleTalkPresent          = 0x00000004,
    gestaltOpenTptAppleTalkLoaded           = 0x00000008,
    gestaltOpenTptTCPPresent                = 0x00000010,
    gestaltOpenTptTCPLoaded                 = 0x00000020,
    gestaltOpenTptIPXSPXPresent             = 0x00000040,
    gestaltOpenTptIPXSPXLoaded              = 0x00000080
};
```

For more information about the `Gestalt` function, see *Inside Macintosh: Operating System Utilities.*

Port-Related Events

There are several port-related events that Open Transport can send to an application that is registered as an Open Transport client. Note that if your application is not registered as a client, Open Transport cannot send it these events.

```
enum {
    kOTPortDisabled         = (OTEventCode)0x25000001,
    kOTPortEnabled          = (OTEventCode)0x25000002,
    kOTYieldPortRequest     = (OTEventCode)0x25000005,
    kOTNewPortRegistered    = (OTEventCode)0x25000006,
};
```

Constant descriptions

`kOTPortDisabled` A port has gone off line, as when the user removes a PCMCIA card while the computer is running. The

`OTResult` parameter gives the specific reason, if known, and the `cookie` parameter is the port reference of the port that went off line. A port going off line 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 that had previously been disabled is now reenabled, as when the user reinserts a previously removed PCMCIA card while the computer is running. The `cookie` parameter is the port reference of the port that is now enabled.

`kOTYieldPortRequest`
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` structure.

`kOTNewPortRegistered`
A new port has been registered with Open Transport, as when the user inserts a new PCMCIA card. The `cookie` parameter is the port reference of the new port. Your provider receives this event the first time a new port is enabled. Subsequently, if a port is reenabled after being disabled, you receive the `kOTPortEnabled` event instead.

The Configuration Structure

Open Transport open-provider functions take as a parameter a pointer to a configuration structure that specifies the configuration of a provider. For example, the configuration structure of an endpoint specifies which protocol modules the endpoint uses. The contents of the configuration structure are private. To create a configuration structure and obtain a pointer to it, you call the `OTCreateConfiguration` function (page 6-27).

The configuration structure is defined by the `OTConfiguration` data type.

```
struct OTConfiguration;
typedef struct OTConfiguration OTConfiguration;
```

The Port Structure

Open Transport uses a port structure to describe a port's characteristics, such as its port name, its child ports, whether it is active or disabled, whether it is private or sharable, and the kind of framing it can use.

The port structure is defined by the `OTPortRecord` data type.

```
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];
};
```

Field descriptions

`fRef` The port reference; a 32-bit value encoding the port's device type, bus type, slot number, and multiport identifier. For more details, see the next section, "The Port Reference."

`fPortFlags` Flags describing the port's status. Only 1 bit can be set at a time. If no bits are set, the port is currently inactive—that is, it is not in use at this time.

Flag	Value	Description
`kOTPortIsActive`	`0x00000001`	The port is in use.
`kOTPortIsDisabled`	`0x00000002`	The port may or may not be in use, but no other client can use it.
`kOTPortIsUnavailable`	`0x00000004`	The port is not available for use.

`fInfoFlags` Flags providing additional information about the port.

Flag	Value	Description
`kOTPortIsDLPI`	`0x00000001`	The port is a DLPI Streams module.
`kOTPortIsTPI`	`0x00000002`	The port is a TPI Streams module.
`kOTPortCanYield`	`0x00000004`	The port can yield when requested.
`kOTPortIsSystemRegistered`	`0x00004000`	The port is registered in the system registry.
`kOTPortIsPrivate`	`0x00008000`	The port is a private port.
`kOTPortIsAlias`	`0x80000000`	The port is an alias for another port.

`fCapabilities` Flags indicating the type of framing capability that a port has. If the port can handle only one type of framing, this field is 0. For example, Ethernet framing uses the following values:

Flag	Value	Description
`kOTFramingEthernet`	`0x01`	The port can use standard Ethernet framing.
`kOTFramingEthernetIPX`	`0x02`	The port can use IPX Ethernet framing.
`kOTFraming8023`	`0x04`	The port can use 802.3 Ethernet framing.
`kOTFraming8022`	`0x08`	The port can use 802.2 Ethernet framing.

`fNumChildPorts` The number of child ports associated with this port.

`fChildPorts` An array of the port references for the child ports associated with this port.

`fPortName` A unique name for this port. The port name is a zero-terminated string that can have a maximum length as indicated by the constant `kMaxProviderNameSize`.

fModuleName	The name of the actual Streams module that implements the driver for this port. Open Transport uses this name internally; you do not use this name.
fSlotID	An 8-byte identifier for a port's slot that contains a 7-byte character string plus a zero for termination. This identifier is typically available for PCI cards.
fResourceInfo	A zero-terminated string that describes a shared library that can handle configuration information for the device. This field contains an identifier that allows Open Transport to access auxiliary information about your driver (Open Transport creates shared library IDs from this string to be able to find these extra shared libraries). This string should either be unique to your driver or should be set to a NULL string.
fReserved	Reserved.

The Port Reference

Several Open Transport port information functions take as a parameter a pointer to a port reference, which is a 32-bit value OTPortRef) that contains a port's device and bus type, its slot number, and information to distinguish among several devices on a single slot. The contents of the port reference are private. To create a port reference and obtain a pointer to it, you call the OTCreatePortRef function (page 6-36).

The port reference is defined by the OTPortRef data type.

```
typedef UInt32 OTPortRef;
```

You can use the OTCreatePortRef function (page 6-36) to create a port reference, and the port reference is also a field in the port structure returned by the port information functions: OTGetIndexedPort (page 6-33), OTFindPort (page 6-34), and OTFindPortByRef (page 6-35). To extract information from the port reference, you need to use the functions OTGetDeviceTypeFromPortRef (page 6-38), OTGetBusTypeFromPortRef (page 6-39), and OTGetSlotFromPortRef (page 6-40).

This section lists the possible values for the device type and bus type.

Note
Do not arbitrarily add new device types. Please contact Developer Support at Apple Computer, Inc. to obtain a new, unique device type. ◆

Possible hardware device types are given in the following enumeration:

```
enum {
    kOTNoDeviceType                 = 0,
    kOTADEVDevice                   = 1,
    kOTMDEVDevice                   = 2,
    kOTLocalTalkDevice              = 3,
    kOTIRTalkDevice                 = 4,
    kOTTokenRingDevice              = 5,
    kOTISDNDevice                   = 6,
    kOTATMDevice                    = 7,
    kOTSMDSDevice                   = 8,
    kOTSerialDevice                 = 9,
    kOTEthernetDevice               = 10,
    kOTSLIPDevice                   = 11,
    kOTPPPDevice                    = 12,
    kOTModemDevice                  = 13,
    kOTFastEthernetDevice           = 14,
    kOTFDDIDevice                   = 15,
    kOTATMLANDevice                 = 16,
    kOTATMSNAPDevice                = 17,
    kOTPseudoDevice                 = 1023,
    kOTLastDeviceIndex              = 1022,
    kOTLastSlotNumber               = 255,
    kOTLastOtherNumber              = 255
};
```

Constant descriptions

`kOTNoDeviceType` The port's device type is not specified. This value is illegal.

`kOTADEVDevice` The port is specified as an ADEV device, which is a pseudodevice used by AppleTalk.

`kOTMDEVDevice` The port is specified as an MDEV device, which is a pseudodevice used by TCP.

`kOTLocalTalkDevice`
The port is specified as a LocalTalk device.

`kOTIRTalkDevice`	The port is specified as an IRTalk device.
`kOTTokenRingDevice`	The port is specified as a token ring device.
`kOTISDNDevice`	The port is specified as an ISDN device.
`kOTATMDevice`	The port is specified as an ATM device.
`kOTSMDSDevice`	The port is specified as a SMDS device.
`kOTSerialDevice`	The port is specified as a serial device.
`kOTEthernetDevice`	The port is specified as an Ethernet device.
`kOTSLIPDevice`	The port is specified as a SLIP pseudodevice.
`kOTPPPDevice`	The port is specified as a PPP pseudodevice.
`kOTModemDevice`	The port is specified as a modem pseudodevice.
`kOTFastEthernetDevice`	The port is specified as an 100 MB Ethernet device.
`kOTFDDIDevice`	The port is specified as a FDDI device.
`kOTATMLANDevice`	The port is specified as an ATM pseudodevice simulating a LAN device, using ATMLAN emulation.
`kOTATMSNAPDevice`	The port is specified as an ATM pseudodevice simulating a SNAP device.
`kOTPseudoDevice`	The port is designated as a pseudodevice.
`kOTLastDeviceIndex`	The maximum number of different device types that a port can support.
`kOTLastSlotNumber`	The highest physical slot number a port can use.
`kOTLastOtherNumber`	The maximum number of ports a single slot can support.

Possible bus types are given in the following enumeration:

```
enum{
        kOTUnknownBusPort           = 0,
        kOTMotherboardBus           = 1,
        kOTNuBus                    = 2,
        kOTPCIBus                   = 3,
        kOTGeoPort                  = 4,
```

```
        kOTPCMCIABus                = 5,
        kOTLastBusIndex             = 15
    };
```

Constant descriptions

`kOTUnknownBusPort`
The port's bus type is not a known type.

`kOTMotherboardBus`
The port is on the motherboard bus.

`kOTNuBus` The port is on a NuBus bus.

`kOTPCIBus` The port is on a PCI bus.

`kOTGeoPort` The port is a GeoPort device.

`kOTPCMCIABus` The port is on a PCMCIA bus.

`kOTLastBusIndex` The maximum number of different bus types that the port can support.

The Client List Structure

When you issue a yield port request with the `OTYieldPortRequest` function, the `buffer` parameter points to a client list structure that identifies the clients that denied the request.

The client list structure is defined by the `OTClientList` data type.

```
struct OTClientList
{
    size_t  fNumClients;
    UInt8   fBuffer[4];
};
typedef struct OTClientList OTClientList;
```

Field descriptions

`fNumClients` The number of clients in the `fBuffer` array, normally 1.

`fBuffer` An array of concatenated Pascal strings enumerating the name of each client that rejected the request—that is, the names under which the clients registered themselves as an Open Transport clients.

The Port Close Structure

When you are using a port that another client wishes to use, the other client can use the `OTYieldPortRequest` function to ask you to yield the port. If you are registered as a client of Open Transport, you receive a `kOTYieldPortRequest` event, whose `cookie` parameter is a pointer to a port close structure. You can use this structure to deny or accept the yield request.

The port close structure is defined by the `OTPortCloseStruct` data type.

```
struct OTPortCloseStruct
{
OTPortRef       fPortRef;
ProviderRef     fTheProvider;
OSStatus        fDenyReason;
};
typedef struct OTPortCloseStruct OTPortCloseStruct;
```

Field descriptions

`fPortRef`	The port requested to be closed.
`fTheProvider`	The provider that is currently using the port.
`fDenyReason`	A value that you can leave untouched to accept the yield request. To deny the request, change this value to a negative error code corresponding to the reason for your denial (normally you use the `kOTUserRequestedErr` error).

Currently, this callback is only used for serial ports, but it is applicable to any hardware device that cannot share a port with multiple clients. If the provider is passively listening—that is, bound with a queue length (`qlen`) greater than 0, and you are willing to yield, you need do nothing. If, however, you are actively connected and you are willing to yield the port, you must issue a synchronous `OTSndDisconnect` in order to let the port go.

Functions

This section describes the functions you need to initialize Open Transport, configure providers, and specify ports.

Initializing and Closing Open Transport

Open Transport provides three functions that you can use to initialize and close Open Transport.

InitOpenTransport

Initializes all of the Open Transport software modules.

C INTERFACE

```
OSStatus InitOpenTransport(void);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

The `InitOpenTransport` function initializes all the Open Transport software modules. Call this function before using other Open Transport functions. A return value other than `kOTNoError` indicates that the Open Transport software is not installed.

SPECIAL CONSIDERATIONS

If your program uses the Apple Shared Library Manager (ASLM), you must call the `InitLibraryManager` function to initialize it before calling the `InitOpenTransport` function.

SEE ALSO

To find out whether the Open Transport software is available, use the `Gestalt` function, which is described in *Inside Macintosh: Operating System Utilities* and whose returned response bits are described in the section "The Gestalt Selector and Response Bits" (page 6-15).

To initialize only the Open Transport modules that handle ports, use the `InitOpenTransportUtilities` function (page 6-25).

To initialize ASLM, use the `InitLibraryManager` function, described in the *Apple Shared Library Manager Developer's Guide*.

For a set of steps to follow when initializing Open Transport, see the section "Initializing Open Transport" (page 6-9).

To close the Open Transport software, use the `CloseOpenTransport` function (page 6-26).

InitOpenTransportUtilities

Initializes only that part of the Open Transport software that handles ports.

C INTERFACE

```
OSStatus InitOpenTransportUtilities(void);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

The `InitOpenTransportUtilities` function initializes only that portion of the Open Transport software that handles ports. Call this function before using other Open Transport functions. A return value other than `kOTNoError` indicates that the Open Transport software is not installed.

SPECIAL CONSIDERATIONS

If your program uses the Apple Shared Library Manager (ASLM), you must call the `InitLibraryManager` function to initialize it before calling the `InitOpenTransportUtilities` function.

SEE ALSO

To open and load all of the Open Transport software, use the `InitOpenTransport` function (page 6-24).

To find out whether the Open Transport software is installed, use the `Gestalt` function, which is described in *Inside Macintosh: Operating System Utilities* and whose returned response bits are described in the section "The Gestalt Selector and Response Bits" (page 6-15).

For a set of steps to follow when initializing Open Transport, see the section "Initializing Open Transport" (page 6-9).

To close the Open Transport software, use the `CloseOpenTransport` function (page 6-26).

CloseOpenTransport

Shuts down the Open Transport software when you are finished using it.

C INTERFACE

```
void CloseOpenTransport(void);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

The `CloseOpenTransport` function closes the Open Transport software, which tells Open Transport that your client has finished using it. Stand-alone code segments must use this function before they unload from memory.

When applications finish using Open Transport, they have the option of using this function to unload the Open Transport software modules without stopping execution if they have other tasks to perform that do not require Open Transport; otherwise, applications don't need to use this function.

SPECIAL CONSIDERATIONS

If your client uses the Apple Shared Library Manager, you must call the `CleanupLibraryManager` function before calling the `CloseOpenTransport` function.

System software cannot unload Open Transport until the last software module on your computer that called the `InitOpenTransport` or `InitOpenTransportUtilities` function has also called the `CloseOpenTransport` function.

If your client is not an application, you must be sure to call the `CloseOpenTransport` function before unloading from memory.

SEE ALSO

To initialize all of the Open Transport software, call the `InitOpenTransport` function (page 6-24).

To initialize only the port-handling part of Open Transport, call the `InitOpenTransportUtilities` function (page 6-25).

For more information about initializing Open Transport, see the section "Initializing Open Transport" (page 6-9).

Creating, Cloning, and Removing a Configuration Structure

Open Transport provides some functions that you can use to create, clone, and destroy a provider configuration structure. When you finish using a provider of any type, always call the `OTCloseProvider` function to close and delete the provider.

OTCreateConfiguration

Creates a structure defining a provider's configuration.

C INTERFACE

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

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`path` A pointer to a character string describing the provider.

DESCRIPTION

The `OTCreateConfiguration` function creates a configuration structure that defines the software modules, hardware ports, and options that Open Transport is to use when you call a function to open a provider. This is a private structure, defined by the `OTConfiguration` data type (page 6-16). To create one, you use the `path` parameter to pass the `OTCreateConfiguration` function a string describing the provider service desired.

The simplest possible value of the `path` parameter is a single protocol module name of the highest-level protocol you want to use; for example, "tcp." If you do not specify a complete communications path, the Open Transport software uses default settings to construct the rest of the path. For example, if you specify "adsp" for the `path` parameter, Open Transport defaults to using the AppleTalk DataStream Protocol (ADSP) protocol module layered above the Datagram Delivery Protocol (DDP) protocol module and with LocalTalk on the default port, which is the printer port.

If you want to identify a particular port in the configuration string, you use the port name to do so (described in the section "About Port Information," beginning on page 6-5). More typically, however, you leave this value blank—for example, using a string with only "adsp" or "adsp, ddp," which configures the provider with whatever port is specified in the control panel.

To specify more than one protocol module, separate the module names with commas. You can also specify values for options by putting them in parentheses after the protocol name; for example, "adsp, ddp (Checksum=1)" specifies that ADSP is to run on top of DDP and that the checksum option is enabled.

If Open Transport cannot parse the list that you pass in the `path` parameter, the `OTCreateConfiguration` function returns `((OTConfiguration*)-1L)`. If there is insufficient memory to create an `OTConfiguration` structure, the `OTCreateConfiguration` function returns `NULL`.

The `OTCreateConfiguration` function returns a pointer to the configuration structure it creates. You pass this pointer as a parameter to the open-provider functions such as the `OTOpenEndpoint` or `OTOpenMapper` functions (discussed in the chapters "Endpoints" and Mappers" in this book).

SPECIAL CONSIDERATIONS

Functions that open providers delete the `OTConfiguration` structure that they use, so you need to use the `OTCloneConfiguration` function to clone a configuration structure if you want to open multiple providers with the same configuration.

SEE ALSO

For more information about creating configuration structures, see the sections "About Provider Configurations" (page 6-3) and "Creating and Reusing Provider Configurations" (page 6-10).

To copy an `OTConfiguration` structure, call the `OTCloneConfiguration` function (page 6-30).

To delete an `OTConfiguration` structure, call the `OTDestroyConfiguration` function (page 6-31).

You can use the functions in "Obtaining Port Information" (page 6-11) to get the names of any or all of the hardware ports on the system.

To create and open an endpoint, call the `OTOpenEndpoint` function or the `OTAsyncOpenEndpoint` function, both described in the chapter "Endpoints" in this book.

To create and open a mapper, call the `OTOpenMapper` function or the `OTAsyncOpenMapper` function, both described in the chapter "Mappers" in this book.

To create and open an AppleTalk service provider, call the `OTOpenAppleTalkServices` function or the `OTAsyncOpenAppleTalkServices` function, both described in the chapter "AppleTalk Services" in this book.

To create and open a TCP/IP service provider, call the `OTOpenInternetServices` function or the `OTAsyncOpenInternetServices` function, both described in the chapter "TCP/IP Services" in this book.

OTCloneConfiguration

Copies an `OTConfiguration` structure.

C INTERFACE

```
OTConfiguration* OTCloneConfiguration(OTConfiguration* cfig);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`cfig`	A pointer to the `OTConfiguration` structure that you want to copy.

DESCRIPTION

The `OTCloneConfiguration` function copies the `OTConfiguration` structure that you specify in the `cfig` parameter and returns a pointer to the copy. Because the internal format of an `OTConfiguration` structure is private, you must use the `OTCloneConfiguration` function to obtain two identical structures. For example, you can use this function when another application passes you a configuration structure that you want to reuse but for which you do not have the original configuration string. By cloning the structure, you have access to an additional copy of the configuration even without knowing its configuration string.

SEE ALSO

For more information about creating configuration structures, see the sections "About Provider Configurations" (page 6-3) and "Creating and Reusing Provider Configurations" (page 6-10).

To create an `OTConfiguration` structure, call the `OTCreateConfiguration` function (page 6-27).

To delete an `OTConfiguration` structure, call the `OTDestroyConfiguration` function (page 6-31).

OTDestroyConfiguration

Deletes an `OTConfiguration` structure.

C INTERFACE

```
void OTDestroyConfiguration(OTConfiguration* cfig);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`cfig`	A pointer to the `OTConfiguration` structure that you want to delete.

DESCRIPTION

The `OTDestroyConfiguration` function deletes the `OTConfiguration` structure that you specify in the `cfig` parameter and releases all associated memory.

SPECIAL CONSIDERATIONS

Functions that open providers delete the `OTConfiguration` structure that they use. For this reason, most applications need not call the `OTDestroyConfiguration` function. You should call the `OTDestroyConfiguration` function only to delete an `OTConfiguration` structure not used to open a provider.

SEE ALSO

For more information about creating configuration structures, see the sections "About Provider Configurations" (page 6-3) and "Creating and Reusing Provider Configurations" (page 6-10).

To create an `OTConfiguration` structure, call the `OTCreateConfiguration` function (page 6-27).

To copy an `OTConfiguration` structure, call the `OTCloneConfiguration` function (page 6-30).

Getting Information About Ports

Open Transport provides several functions that obtain information about ports available to your system.

OTGetProviderPortRef

Identifies the port associated with a given provider.

C INTERFACE

```
OTPortRef OTGetProviderPortRef(ProviderRef ref);
```

C++ INTERFACES

```
OTPortRef TProvider::GetOTPortRef();
```

PARAMETERS

`ref` The provider for which you wish to obtain the port.

DESCRIPTION

The `OTGetProviderPortRef` function returns the port reference for the provider identified by the `ref` parameter. If the function returns 0, then either no port was associated with this provider or there were multiple associated ports and Open Transport did not know which to return.

OTGetIndexedPort

Iterates through the ports available on your computer.

C INTERFACE

```
Boolean OTGetIndexedPort(OTPortRecord* record,
                         size_t index);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`record`	A pointer to a port structure that contains information about a specific port on your computer.
`index`	An index number.

DESCRIPTION

The `OTGetIndexedPort` function returns information about the ports available on your local system. To iterate through all the ports on your computer, call the function repeatedly, incrementing the `index` parameter each time (starting with 0) until the function returns `false`. Each time the function returns `true`, it fills in the port structure that you provide with information about a specific port. You can use this information, for example, when specifying a provider configuration string for the `OTCreateConfiguration` function.

You must allocate the port structure; the function fills this structure with information about the port indicated by the `index` parameter. If the function returns `false`, the contents of the structure are not significant.

SEE ALSO

For information about finding ports, see the sections “About Port Information” (page 6-5) and “Obtaining Port Information” (page 6-11).

The port structure is described in “The Port Structure” (page 6-17).

OTFindPort

Obtains information about a port that corresponds to a given port name.

C INTERFACE

```
Boolean OTFindPort  (OTPortRecord* record,
                     const char* portName);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`record`	A pointer to a port structure that contains information about the port you specified with the `portName` parameter.
`portName`	The name of the port about which you want information.

DESCRIPTION

The `OTFindPort` function returns information about a port that corresponds to a given port name. Each port in a system has a unique port name, which you can obtain through a previous call or set of calls to the `OTGetIndexedPort` function.

You must allocate the port structure; the function fills this structure with information about the port indicated by the `portName` parameter. If the function returns `false`, the contents of the structure are not significant.

SEE ALSO

For information about finding ports, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

You can use the `OTGetIndexedPort` function (page 6-33) to get port information by iterating through all available ports.

The port structure is described in "The Port Structure" (page 6-17).

OTFindPortByRef

Obtains information about a port that corresponds to its given port reference.

C INTERFACE

```
Boolean OTFindPortByRef(OTPortRecord* record,
                        OTPortRef ref);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`record`	A pointer to a port structure that contains information about the port you specified with the `ref` parameter.
`ref`	The port reference of the port about which you want information.

DESCRIPTION

The `OTFindPortByRef` function returns information about a port identified by its port reference. A port reference is a 32-bit value that describes a port's hardware characteristics: its bus and device type, its physical slot number, and, where applicable, its multiport identifier. This identifier differentiates between multiple hardware ports on a given slot.

You must allocate the port structure; the function fills this structure with information about the port indicated by the `ref` parameter. If the function returns `false`, the contents of the structure are not significant.

SEE ALSO

For information about finding ports and obtaining a port reference, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

You can use the `OTCreatePortRef` function (page 6-36) to create a port reference.

The port structure is described in "The Port Structure" (page 6-17).

OTCreatePortRef

Creates a port reference that describes a port's hardware characteristics.

C INTERFACE

```
OTPortRef OTCreatePortRef(UInt8 busType,
                          UInt16 devType,
                          UInt16 slot,
                          UInt16 other);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`busType`	The type of bus to which the hardware port is connected; for example, a NuBus or PCI bus. See "The Port Reference" (page 6-19) for possible values for this parameter.
`devType`	The type of hardware device connected to the port, such as LocalTalk or Ethernet. See "The Port Reference" (page 6-19) for possible values for this parameter.
`slot`	The number of the physical slot containing the device.
`other`	The port's multiport identifier—that is, a numeric value that distinguishes between ports when more than one hardware port is connected to a given slot.

DESCRIPTION

The `OTCreatePortRef` function creates a port reference structure, which is a 32-bit value that describes a port's hardware characteristics: its device and bus type, its physical slot number, and, where applicable, its multiport identifie.

Once you have created a port reference, you can use the `OTFindPortByRef` function to find a specific port with that particular set of characteristics.

To create a port reference, you use the `OTCreatePortRef` function. You must know all the port's hardware characteristics: its device and bus type, its slot number, and its multiport identifier (if it has one). You cannot use wildcards to fill in any element you don't know, although you can use a device type of 0 to allow matches on every kind of device type (following the zero-matches-everything rule). Possible device and bus types are described in the section "The Port Reference" (page 6-19).

To create a port reference for a pseudodevice, use 0 as the value for the bus type, slot number, and multiport identifier, and use the constant `kOTPseudoDevice` for the device type.

Open Transport has predefined variants of the `OTCreatePortRef` function for the most commonly used hardware devices, such as the NuBus, PCI, and PCMCIA devices. These three variants are listed here:

```
#define OTCreateNuBusPortRef(devType, slot, other)\
        OTCreatePortRef(kOTNuBus, devType, slot, other)

#define OTCreatePCIPortRef(devType, slot, other)\
        OTCreatePortRef(kOTPCIBus, devType, slot, other)

#define OTCreatePCMCIAPortRef(devType, slot, other)\
        OTCreatePortRef(kOTPCMCIABus, devType, slot, other)
```

Once you have identified the port structure you want, you can access the information in its port reference, by using the `OTGetDeviceTypeFromPortRef`, `OTGetBusTypeFromPortRef`, and `OTGetSlotFromPortRef` functions.

SPECIAL CONSIDERATIONS

The slot numbers are physical; that is, they are the slot numbers returned by the Slot Manager and not the slots seen in various network configuration applications. Physical slot numbers depend on the type of card installed. For example, NuBus cards number their slots 9–13, which appear in the AppleTalk or TCP control panels as slots 1–5.

SEE ALSO

For information about finding ports and obtaining a port reference, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

See "The Port Reference" (page 6-19) for possible values for the bus type and device type.

To extract information from a port reference, use the `OTGetDeviceTypeFromPortRef` function (page 6-38),the `OTGetBusTypeFromPortRef` function (page 6-39), and the `OTGetSlotFromPortRef` function (page 6-40).

The port structure is described in "The Port Structure" (page 6-17).

OTGetDeviceTypeFromPortRef

Extracts the value of the hardware device type from a port reference.

C INTERFACE

```
UInt16 OTGetDeviceTypeFromPortRef(OTPortRef ref);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`ref`	The port reference from which you wish to extract the device type.

DESCRIPTION

The `OTGetDeviceTypeFromPortRef` function extracts the device type value from a port reference with unknown hardware values. You can obtain such a port reference when another application passes one to you or when you use the `OTGetIndexedPort` function to access a port structure into which another application has put its own port reference.

SEE ALSO

For information about finding ports and obtaining a port reference, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

The possible return values for the `OTGetDeviceTypeFromPortRef` function are listed in "The Port Reference" (page 6-19).

You can use the `OTGetBusTypeFromPortRef` function (page 6-39) and the `OTGetSlotFromPortRef` function (page 6-40) to get the bus type and slot number information from the port reference.

You can use the `OTCreatePortRef` function (page 6-36) to create a port reference.

Port references are returned by the `OTGetIndexedPort` function (page 6-33), the `OTFindPort` function (page 6-34), and the `OTFindPortByRef` function (page 6-35).

The port structure is described in "The Port Structure" (page 6-17).

OTGetBusTypeFromPortRef

Extracts the value of the bus type from a port reference.

C INTERFACE

```
UInt16 OTGetBusTypeFromPortRef(OTPortRef ref);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`ref` The port reference from which you wish to extract the bus type.

DESCRIPTION

The `OTGetBusTypeFromPortRef` function extracts the bus type value from a port reference with unknown hardware values. You can obtain such a port reference when another application passes one to you or when you use the `OTGetIndexedPort` function to access a port structure into which another application has put its own port reference.

SEE ALSO

For information about finding ports and obtaining a port reference, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

The possible return values for the `OTGetBusTypeFromPortRef` function are listed in "The Port Reference" (page 6-19).

You can use the `OTGetDeviceTypeFromPortRef` function (page 6-38) and the `OTGetSlotFromPortRef` function (page 6-40) to get device type and slot number information from the port reference.

You can use the `OTCreatePortRef` function (page 6-36) to create a port reference.

Port references are returned by the `OTGetIndexedPort` function (page 6-33), the `OTFindPort` function (page 6-34), and the `OTFindPortByRef` function (page 6-35).

The port structure is described in "The Port Structure" (page 6-17).

OTGetSlotFromPortRef

Extracts slot information from a port reference.

C INTERFACE

```
UInt16 OTGetSlotFromPortRef(OTPortRef ref,
                            UInt16* other);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`ref`	The port reference from which you wish to extract the slot number.
`other`	A pointer to a 16-bit buffer you provide into which the function places a value that distinguishes between ports when more than one hardware port is connected to a given slot. Specify `NULL` for this parameter if you do not want the function to return this information.

DESCRIPTION

The `OTGetSlotFromPortRef` function extracts slot information from a port reference with unknown hardware values. You can obtain such a port reference when another application passes one to you or when you use the `OTGetIndexedPort` function to access a port structure into which another application has put its own port reference.

Note that the slot numbers are physical; that is, they are the slot numbers returned by the Slot Manager and not the slots seen in various network configuration applications. Physical slot numbers depend on the type of card installed. For example, NuBus cards number their slots 9–13, which appear in the AppleTalk or TCP control panels as slots 1–5.

SEE ALSO

For information about finding ports and obtaining a port reference, see the sections "About Port Information" (page 6-5) and "Obtaining Port Information" (page 6-11).

You can use the `OTGetDeviceTypeFromPortRef` function (page 6-38) and the `OTGetBusTypeFromPortRef` function (page 6-39) to get device type and bus type information from the port reference.

You can use the `OTCreatePortRef` function (page 6-36) to create a port reference.

Port references are returned by the `OTGetIndexedPort` function (page 6-33), the `OTFindPort` function (page 6-34), and the `OTFindPortByRef` function (page 6-35).

The port structure is described in "The Port Structure" (page 6-17) and the port reference is described in "The Port Reference" (page 6-19).

Requesting a Port to Yield Ownership

Open Transport allows you to request that the current owner of a port yield the use of the port to you.

OTYieldPortRequest

Requests that a port be yielded.

C INTERFACE

```
OSStatus OTYieldPortRequest(ProviderRef provider, OTPortRef ref,
                    OTClientList* buffer, size_t size);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`provider`	The provider reference for the provider that wants to use the port. This provider (normally an endpoint) must be open on the requested port. You cannot use this provider while the yield request is being processed.
`ref`	The port reference for the port you wish to use.
`buffer`	If the request is denied, on output this contains a pointer to a client list structure, giving the names of all clients that rejected the request (normally only one). You can use a value of `NULL` to force the client to yield the port; this only works if the provider is passively listening on the port. Use a value of `(void*)-1L` to cancel the yield request.
`size`	The size of the buffer, including the `fNumClients` field in the client list structure.

DESCRIPTION

The `OTYieldPortRequest` function requests the current owner of a port (normally a serial port or modem) to yield ownership of it. Open Transport sends the `kOTYieldPortRequest` event to each provider of any registered clients for that port for acceptance or denial. If the owner has not registered as a client of Open Transport, no event can be sent and acceptance is implicit.

If the function returns an error, the request could not be completed. This could be because the current owner refused to yield, in which case the error code provides the reason for the denial.This also could be because there is not enough memory (`kENOMEMErr`), the port does not support yielding (`kOTNotSupportedErr`), the provider does not use the port or the port does not exist (`kOTBadReferenceErr`), or because the current client is already connected (`kENOENTErr`).

Once the yield port request returns with an `kOTNoError` result code (or `kENOENTErr`, if the provider does not currently have a client), you can attempt to use the port. Normally you do this by binding with a queue length (`qlen`) greater than 0, or by connecting. If you do not use the port or cancel the yield request within 10 seconds, the port automatically stops being available for your use and reverts to its original owner.

In the case of forcing the client to yield, if the function returns `kOTNoError`, then the port has been yielded, and you can use it. Note that you can only force a port to yield if its current client is passively listening; it cannot be yielded if a connection is in progress.

SPECIAL CONSIDERATIONS

The `OTYieldPortRequest` function is available only to PowerPC-native clients; there is no mixed-mode glue for the function, so you must build your application FAT in order to use it.

SEE ALSO

To register as a client, use the `OTRegisterAsClient` function (page 6-44).

The client list structure is described in “The Client List Structure” (page 6-22).

Registering as a Client

Open Transport provides functions that you can use to register or unregister your application as a client of Open Transport.

OTRegisterAsClient

Registers your application as a client of Open Transport and gives Open Transport a notifier function it can use to send you events.

C INTERFACE

```
OSStatus OTRegisterAsClient(OTClientName name,
                            OTNotifyProcPtr proc)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`name`	A pointer to the user-readable name you want Open Transport to use for your application.
`proc`	A pointer to the notifier function you want Open Transport to use for sending events to your application.

DESCRIPTION

The `OTRegisterAsClient` function registers your application as an Open Transport client. This function provides Open Transport with a pointer to your notifier function that it can call when port transition events occur. It also provides a user-readable name Open Transport can use when it delivers messages about these events to the user. This function is optional; if you do not want to receive these events, you do not have to call this function.

SEE ALSO

For more information about registering as a client, see the section "Registering as an Open Transport Client," beginning on page 6-13.

To unregister yourself as an Open Transport client, use the `OTUnregisterAsClient` function (page 6-45).

OTUnregisterAsClient

Removes your application as a client of Open Transport.

C INTERFACE

```
OSStatus OTUnregisterAsClient(void)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

SPECIAL CONSIDERATIONS

If you do not call the `OTUnregisterAsClient` function, the `CloseOpenTransport` function calls it for you automatically when it executes.

SEE ALSO

For more information about registering as a client, see the section "Registering as an Open Transport Client," beginning on page 6-13.

To register yourself as an Open Transport client, use the `OTRegisterAsClient` function (page 6-44).

Process Management

Contents

This chapter describes how you can use Open Transport functions to handle system tasks and deferred tasks in existing System 7 code. You cannot call most Open Transport functions at interrupt time, so you need to ensure that your code makes such calls during system or deferred task time.

This chapter describes

- checking whether you can make a synchronous call
- handling system and deferred tasks
- calling permitted Open Transport functions at interrupt time
- idling or delaying processing on your computer

To use this chapter, you need to be familiar with System 7 system tasks, deferred tasks, and interrupts in general. For information about system tasks, read the information about the `SystemTask` function in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials,* and for information about interrupts and deferred tasks, read the chapters "Introduction to Processes and Tasks" and "Deferred Task Manager" in *Inside Macintosh: Processes*.

This chapter begins with an introduction to Open Transport task processing. Then it discusses System 7 system tasks and deferred tasks in more detail and explains the process callback functions you use to create these tasks. Mac Os 8 task processing will be described in later documentation.

About Task Processing in Open Transport

If you are writing an application, the Open Transport system task and deferred task functions are optional. You don't need the system task functions because your code executes within a normal event loop, referred to as *system task time*, so you can call Open Transport functions as part of your application's normal processing. You don't need the deferred task functions because when you execute your Open Transport code asynchronously, much of the processing takes place in the notifier functions, which can call Open Transport functions because notifiers execute at deferred task time.

However, you may wish to use the Open Transport system task functions for some of your application's processing because the Open Transport functions provide an efficient way to streamline your main event loop. For example, you

can avoid the complication of handling some of your memory allocation during your main event loop; instead, you can schedule a system task to obtain memory at certain times or on a periodic basis.

If you are writing code that isn't an application (for example, a system extension or a code resource), you probably need to use the system and deferred task functions available in Open Transport to get processing time to handle such tasks as allocating memory or accessing disk space.

If you currently use interrupts in your code or if you want to call Open Transport from within an interrupt function such as a Time Manager, or Vertical Retrace Manager function, you must use the Open Transport deferred task functions to handle the task as a deferred task.

Open Transport provides several functions to make it easy for you to defer your interrupt operations to deferred task time. The Open Transport deferred task functions provide an alternative to the Deferred Task Manager `DTInstall` function. Using the Open Transport higher-level deferred task functions instead of the `DTInstall` function allows OpenTransport to adapt to changes in the underlying operating system without affecting the client's code.

Using Process Management Functions

Most of the functions in this chapter are concerned with system and deferred tasks. Much of the processing for these two kinds of tasks is similar, so they are covered in the same section. There are two functions for notifying Open Transport that you are entering and leaving interrupt processing, and although these are associated with using deferred tasks, they are covered in a separate section. There is a brief section on the little-used functions for idling and delaying your computer. Finally, there is a section on how to write an application-defined process callback function that you can use when creating system and deferred tasks.

Using System and Deferred Tasks

To use system or deferred tasks with the Open Transport functions, you use a process callback function that defines the task you want executed at a scheduled system or deferred task time. When you call the Open Transport function that creates a system task with the `OTCreateSystemTask` function (or

creates a deferred task with the `OTCreateDeferredTask` function), you pass a pointer to your process callback function so that Open Transport can call it at the specified time. You can also pass user-defined context information at this time. When Open Transport calls back your function, it passes back the context data you supplied when you created the task. For the 68000-family of Macintosh computers, Open Transport also restores the A5 world to what it was when you created the task.

The `OTCreateSystemTask` and `OTCreateDeferredTask` functions allocate a structure that defines the task you want executed. Upon completion, these functions return a reference by which you subsequently refer to the task in other system or deferred task functions.

Once you have created a task, you need to schedule it for execution. To do this, you use the `OTScheduleSystemTask`, the `OTScheduleDeferredTask`, or the `OTScheduleInterruptTask` function. You pass the task reference to the function (the `stCookie` or the `dtCookie` parameter), and Open Transport attempts to schedule the task. If a system task is scheduled successfully, it executes when the `SystemTask` function next executes; and if a deferred task is scheduled successfully, it executes as soon as possible after leaving the interrupt context.

However, because a system task can happen relatively slowly, enough time can elapse between scheduling and execution to let you cancel it before it runs. Deferred tasks happen too quickly to allow time for canceling tasks. If you use the `OTCancelSystemTask` function, you notify Open Transport not to execute the system task at the scheduled time. The reference is still valid, and you can choose to reschedule the task by using the `OTScheduleSystemTask` function again at any time.

You can also choose to reschedule a system or deferred task after it has executed successfully. You do this by using the `OTScheduleSystemTask` or the `OTScheduleDeferredTask` function again at any time. If you choose to reschedule a task, you reuse the same reference to the same task. This means that exactly the same task executes, which is useful for repetitive periodic tasks.

You can choose to destroy a task with the `OTDestroySystemTask` or the `OTDestroyDeferredTask` function. These functions make the task reference invalid and free any resources associated with the task. You can call these functions whenever it is no longer necessary to schedule a task, such as when it has been executed at its scheduled time and you have no plans to reschedule it for later use.

You can call the OTDestroySystemTask function to destroy a system task that is currently scheduled for execution. In this case, Open Transport cancels the system task before proceeding with the task's destruction.

If you try to destroy a scheduled deferred task with the OTDestroyDeferredTask function, the kEAgainErr error can occur. This is a rare situation that can only happen when you try to destroy the task from within an interrupt service routine or within another deferred task.

If you want to use a task after you have destroyed it, you must start from the beginning again by creating a new task with the OTCreateSystemTask or the OTCreateDeferredTask functions.

Entering and Leaving Interrupt Processing

There are some Open Transport functions that you can call at interrupt time, but you must be sure to notify Open Transport that you are doing so and notify Open Transport when you are done. The permitted functions are listed in Table 7-1.

Table 7-1 Open Transport functions you can call at interrupt time

Function	Description
OTCanMakeSyncCall	Checks whether a synchronous call will fail
OTAllocMem	Allocates raw memory
OTFreeMem	Frees the memory allocated with OTAllocMem
OTEnterInterrupt	Prepares for an Open Transport function call during interrupt time
OTLeaveInterrupt	Concludes the processing of an Open Transport function call during interrupt time
OTScheduleSystemTask	Schedules a system task
OTCancelSystemTask	Cancels a system task
OTScheduleDeferredTask	Schedules a deferred task
OTScheduleInterruptTask	Schedules a deferred task

If you are at interrupt time and you want to call an Open Transport function, you must first call the `OTEnterInterrupt` function. You can then call one of the permitted functions. When you are done with calling Open Transport functions at interrupt time, you must call the `OTLeaveInterrupt` function. For example, you could execute these code statements in this sequence:

```
OTEnterInterrupt();
OTScheduleDeferredTask(dtCookie);
OTLeaveInterrupt();
```

The exception to this set of tasks is the `OTScheduleInterruptTask` function. If all you want to do is schedule a deferred task, you can use the `OTScheduleInterruptTask` function, which takes care of the `OTEnterInterrupt` and `OTLeaveInterrupt` functions for you.

▲ **WARNING**
If you try to call an Open Transport function that is not permitted at interrupt time or if you do not use the `OTEnterInterrupt` and `OTLeaveInterrupt` functions, you will either get the `OTBadSyncErr` result code or crash your system, depending on the function you call. ▲

Allocating and Freeing Raw Memory

Open Transport provides two interrupt-safe functions to allow you to allocate memory from the Open Transport memory pool.

You can use the `OTAllocMem` function to obtain raw memory. The memory is allocated from a pool that Open Transport has created on behalf of the client application. You need to use the `OTAllocMem` function to be able to access this memory. You need to use this function if you want to allocate memory within an interrupt. To deallocate this memory, use the `OTFreeMem` function with the pointer returned by the `OTAllocMem` function.

Idling or Delaying Your Computer

There are two little-used functions for idling or delaying your computer: `OTIdle` and `OTDelay`. The former function does not currently offer any practical use beyond providing compatibilty with existing code that already uses an idle function. The latter function is only included for compatibility with the UNIX

`sleep` function and should not be used in your code for any other reason. Be sure not to include in any production code in your products.

Process Management Reference

This section describes the Open Transport functions you need to use to handle system tasks and deferred tasks. It also describes how to enter and leave interrupt time, how to use an idle function, and how to use a delay function that is equivalent to the UNIX `sleep` function.

Functions

This section describes the functions you need to use system and deferred tasks, to enter and leave interrupt time, and to use a delay function.

Checking Synchronous Calls

Open Transport provides the `OTCanMakeSyncCall` function that allows you to check beforehand whether you can call a synchronous function at a given moment.

OTCanMakeSyncCall

Checks whether you can call a synchronous function.

C INTERFACE

```
Boolean OTCanMakeSyncCall()
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

Working With System Tasks

Open Transport provides several functions that allow you to handle system tasks in your code.

OTCreateSystemTask

Allows a function to be executed at the next system task time.

C INTERFACE

```
long OTCreateSystemTask(OTProcessProcPtr proc,
                        void* arg)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`proc`	A pointer to the process callback function you want executed at system task time.
`arg`	A pointer to application-defined data that Open Transport can pass to your callback function. Pass `NULL` if you do not want this data passed. If you are creating more than one of the same kind of task, you can use different values for `arg` to distinguish between the tasks.

DESCRIPTION

The `OTCreateSystemTask` function creates a system task that you can schedule for execution at the next system task time. The task contains a pointer to the process callback function specified by the `proc` parameter. At the next system task time, Open Transport calls your process callback function, passing it the `arg` parameter and, for the 68000-family of Macintosh computers only, restoring

the A5 global world to what it was when you originally called `OTCreateSystemTask`.

This function returns a reference that you can use to identify a task in other system task functions. If the return value is 0, then there is not enough memory to allocate the necessary data.

SEE ALSO

To schedule a task for execution at system task time, call the `OTScheduleSystemTask` function (page 7-10).

To destroy a system task created with the `OTCreateSystemTask` function, call the `OTDestroySystemTask` function (page 7-13).

To cancel a task scheduled for execution at system task time, call the `OTCancelSystemTask` function (page 7-12).

OTScheduleSystemTask

Schedules a task for execution at system task time.

C INTERFACE

```
Boolean OTScheduleSystemTask(long stCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`stCookie` A reference that identifies the task to be scheduled.

DESCRIPTION

The `OTScheduleSystemTask` function schedules for execution at system task time the task associated with the `stCookie` parameter, which is the reference returned by the `OTCreateSystemTask` function.

You can call this function at any time. If you have not yet destroyed a task, you can use this function to reschedule the same task again once it has executed. If you have canceled a task, you can use this function to schedule it again.

If you makes multiple calls to the `OTScheduleSystemTask` function before the task is executed, additional tasks are not scheduled; only one instance of each unique task can only be scheduled at a time.

This function returns `true` if it scheduled the system task successfully, `false` if not. If the function returns `false` and the `stCookie` parameter has a valid value (other than 0), then the task is already scheduled to run. If `stCookie` is invalid (a value of 0), the function returns `false` and does nothing.

SPECIAL CONSIDERATIONS

You can call this Open Transport function at interrupt time, but you must precede it by calling the `OTEnterInterrupt` function and you must follow it by calling the `OTLeaveInterrupt` function.

SEE ALSO

To create a system task, call the `OTCreateSystemTask` function (page 7-9).

To destroy a system task created with the `OTCreateSystemTask` function, call the `OTDestroySystemTask` function (page 7-13).

To cancel a task scheduled for execution at system task time, call the `OTCancelSystemTask` function (page 7-12).

Before making this call from within an interrupt, use the `OTEnterInterrupt` function (page 7-19).

After you have made this call from within an an interrupt, use the `OTLeaveInterrupt` function (page 7-20).

OTCancelSystemTask

Cancels a function you have scheduled to execute at system task time.

C INTERFACE

```
Boolean OTCancelSystemTask(long stCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`stCookie` A reference value that identifies the task to be canceled.

DESCRIPTION

The `OTCancelSystemTask` function cancels a task that was scheduled with the `OTScheduleSystemTask` function to run at system task time. The function returns `true` if the scheduling was canceled. If the function returns `false`, then either the function was not scheduled, or it is too late to cancel it. If the `stCookie` parameter value is invalid (a value of 0), the function returns `false` and does nothing.

SPECIAL CONSIDERATIONS

You can call this Open Transport function at interrupt time, but you must precede it by calling the `OTEnterInterrupt` function and you must follow it by calling the `OTLeaveInterrupt` function.

SEE ALSO

To create a system task, call the `OTCreateSystemTask` function (page 7-9).

To schedule a task for execution at system task time, call the `OTScheduleSystemTask` function (page 7-10).

To destroy a system task created with the `OTCreateSystemTask` function, call the `OTDestroySystemTask` function (page 7-13).

Before making this call from within an interrupt, use the `OTEnterInterrupt` function (page 7-19).

After you have made this call from within an an interrupt, use the `OTLeaveInterrupt` function (page 7-20).

OTDestroySystemTask

Destroys a system task created with the `OTCreateSystemTask` function.

C INTERFACE

```
OSStatus OTDestroySystemTask(long stCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`stCookie` A reference value that identifies the system task to be destroyed.

DESCRIPTION

The `OTDestroySystemTask` function makes the `stCookie` reference invalid and frees any resources allocated to the task when it was created. You can call this function when you no longer need to schedule the system task, such as when it has already been executed at its scheduled time and you have no plans to reschedule it for later use. If `stCookie` is already invalid (a value of 0), the function returns `kOTNoError` and does nothing.

If you try to destroy a task that is still scheduled for execution, the `OTDestroySystemTask` function cancels it first, so that it is no longer scheduled for system task execution, and then destroys it. If the task has already been canceled, the `OTDestroySystemTask` function simply destroys it.

SEE ALSO

To create a system task, call the `OTCreateSystemTask` function (page 7-9).

To schedule a task for execution at system task time, call the `OTScheduleSystemTask` function (page 7-10).

To cancel a task scheduled for execution at system task time, call the `OTCancelSystemTask` function (page 7-12).

Working With Deferred Tasks

Open Transport provides several functions that allow you to handle deferred tasks in your code.

OTCreateDeferredTask

Creates a deferred task that identifies a function to be executed at the next deferred task time.

C INTERFACE

```
long OTCreateDeferredTask(OTProcessProcPtr proc,
                          void* arg)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`proc`	A pointer to the process callback function you want executed at deferred task time.
`arg`	A pointer to application-defined data that Open Transport can pass to your callback function. Pass `NULL` if you do not want this data passed. If you are creating more than one of the same kind of task, you can use different values for `arg` to distinguish between the tasks.

DESCRIPTION

The `OTCreateDeferredTask` function creates a deferred task that you can schedule for execution at the next deferred task time. The task contains a pointer to the process callback function indicated by the `proc` parameter. At the next deferred task time, Open Transport calls your process callback function, passing it the `arg` parameter and, for the 68000-family ofMacintosh computers only, restoring the A5 global world to what it was when you originally called `OTCreateDeferredTask`.

This function returns a reference that you can use to identify a task in other deferred task functions. If the return value is 0, then there is not enough memory to allocate the necessary data.

If you want to call Open Transport from an interrupt, you can use this function (and `OTScheduleDeferredTask`) instead of the standard Deferred Task Manager function `DTInstall` to create a deferred task that you to call permits Open Transport functions. This allows Open Transport to schedule deferred task processing independently of the underlying deferred task mechanism.

SEE ALSO

To schedule a task for execution at deferred task time, call the `OTScheduleDeferredTask` function (page 7-17).

To destroy a task created with the `OTCreateDeferredTask` function, call the `OTDestroyDeferredTask` function (page 7-18).

OTScheduleInterruptTask

Schedules a task for execution at deferred task time.

C INTERFACE

```
Boolean OTScheduleInterruptTask(long dtCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`dtCookie` A reference that identifies the task to be scheduled.

DESCRIPTION

The `OTScheduleInterruptTask` function schedules for execution at the next deferred task time the task associated with the `dtCookie` parameter, which is the reference returned by the `OTCreateDeferredTask` function. This call includes internal calls to the `EnterInterrupt` and the `LeaveInterrupt` functions, so you do not need to make separate calls to those functions as you do with other similar functions.

You can call this function at any time. If you have not yet destroyed a task, you can use this function to reschedule it multiple times.

If you make multiple calls to the `OTScheduleInterruptTask` function before the task is executed, additional tasks are not scheduled; only one instance of each unique task can be scheduled at a time.

This function returns `true` if it scheduled the deferred task successfully, `false` if not. If it returns `false` and the `dtCookie` parameter has a valid value (other than 0), then the task is already scheduled to run. If `dtCookie` is invalid (a value of 0), the function returns `false` and does nothing.

If you want to call Open Transport from an interrupt, you can use this function (and the `OTCreateDeferredTask` function) instead of the standard Deferred Task Manager function `DTInstall` to create a deferred task that permits you to call Open Transport function calls. This allows Open Transport to adapt to changes in the underlying operating system without affecting the client's code.

SPECIAL CONSIDERATIONS

You can call this Open Transport function at interrupt time without calling the `OTEnterInterrupt` function first and the `OTLeaveInterrupt` function afterwards.

SEE ALSO

To create a deferred task for execution at deferred task time, call the `OTCreateDeferredTask` function (page 7-14).

To destroy a task created with the `OTCreateDeferredTask` function, call the `OTDestroyDeferredTask` function (page 7-18).

OTScheduleDeferredTask

Schedules a task for execution at deferred task time.

C INTERFACE

```
Boolean OTScheduleDeferredTask(long dtCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`dtCookie` A reference that identifies the task to be scheduled.

DESCRIPTION

The `OTScheduleDeferredTask` function schedules for execution at the next deferred task time the task associated with the `dtCookie` parameter, which is the reference returned by the `OTCreateDeferredTask` function.

You can call this function at any time. If you have not yet destroyed a task, you can use this function to reschedule the same task more than once.

If you makes multiple calls to the `OTScheduleDeferredTask` function before the task is executed, additional tasks are not scheduled; only one instance of each unique task can only be scheduled at a time.

This function returns `true` if it scheduled the deferred task successfully, `false` if not. If it returns `false` and the `dtCookie` parameter has a valid value (other than 0), then the task is already scheduled to run. If `dtCookie` is invalid (a value of 0), the function returns `false` and does nothing.

If you want to call Open Transport from an interrupt, you can use this function (and the `OTCreateDeferredTask` function) instead of the standard Deferred Task Manager function `DTInstall` to create a deferred task that permits you to call Open Transport function calls. This allows Open Transport to adapt to changes in the underlying operating system without affecting the client's code.

SPECIAL CONSIDERATIONS

You can call this Open Transport function at interrupt time, but you must precede it by calling the `OTEnterInterrupt` function and you must follow it by calling the `OTLeaveInterrupt` function.

SEE ALSO

To create a deferred task for execution at deferred task time, call the `OTCreateDeferredTask` function (page 7-14).

To destroy a task created with the `OTCreateDeferredTask` function, call the `OTDestroyDeferredTask` function (page 7-18).

Before making this call from within an interrupt, use the `OTEnterInterrupt` function (page 7-19).

After you have made this call from within an an interrupt, use the `OTLeaveInterrupt` function (page 7-20).

OTDestroyDeferredTask

Destroys a deferred task created with the `OTCreateDeferredTask` function.

C INTERFACE

```
OSStatus OTDestroyDeferredTask(long dtCookie)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`dtCookie` A reference that identifies the task to be destroyed.

DESCRIPTION

The `OTDestroyDeferredTask` function makes the `dtCookie` reference invalid and frees any resources allocated to the task when it was created. You can call this function at any time when you no longer need to schedule the deferred task object. If `dtCookie` is invalid (a value of 0), the function returns `kOTNoError` and does nothing.

If you try to destroy a deferred task that is still scheduled, the `kEAgainErr` error can occur. This is a rare situation that can only happen when you try to destroy the task from within an interrupt service routine or within another deferred task.

SEE ALSO

To create a deferred task object for execution at deferred task time, call the `OTCreateDeferredTask` function (page 7-14).

To schedule a task for execution at deferred task time, call the `OTScheduleDeferredTask` function (page 7-17).

Entering and Leaving Interrupt Time

Open Transport provides functions that you use to inform Open Transport that you are entering and leaving interrupt time.

OTEnterInterrupt

Notifies Open Transport that you are about to call an Open Transport function from an interrupt.

C INTERFACE

```
void OTEnterInterrupt(void)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

The `OTEnterInterrupt` function informs Open Transport it is at primary interrupt time. This allows Open Transport to more intelligently schedule network activity. You must use this function when you are about to call one of the few Open Transport functions permitted at interrupt time.

SPECIAL CONSIDERATIONS

On the 68000-family of Macintosh computers, you must be sure that your A5 world is set correctly before making this call (that is, having the same value it had when you called the `InitOpenTransport` or the `InitOpenTransportUtilities` function).

You must call the `OTLeaveInterrupt` function before you leave interrupt time.

SEE ALSO

Table 7-1 provides a list of functions that you can call at interrupt time (page 7-6).

Before leaving the interrupt context, use the `OTLeaveInterrupt` function (page 7-20).

For further discussion, see the section "Entering and Leaving Interrupt Processing," beginning on page 7-6.

OTLeaveInterrupt

Notifies Open Transport that you are done calling Open Transport functions from an interrupt.

C INTERFACE

```
void OTLeaveInterrupt(void)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

The `OTLeaveInterrupt` function informs Open Transport that you are no longer calling Open Transport from within an interrupt. You call this function after you call the last Open Transport function you want to call from within the interrupt, but before you return to system task time.

SPECIAL CONSIDERATIONS

Do not make this call without having already called the `OTEnterInterrupt` function.

On the 68000-family of Macintosh computers, you must be sure that your A5 world is set correctly before making this call (that is, you must set it to the same value it had when you called the `InitOpenTransport` or the `InitOpenTransportUtilities` function).

SEE ALSO

To notify Open Transport that you are about to call an Open Transport function from an interrupt, use the `OTEnterInterrupt` function (page 7-19).

For further discussion, see the section "Entering and Leaving Interrupt Processing," beginning on page 7-6.

Allocating and Freeing Memory

Open Transport provides functions to allow you to allocate memory from the Open Transport memory pool.

OTAllocMem

Allocates memory from the Open Transport memory pool.

C INTERFACE

```
void* OTAllocMem(size_t nbytes)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbytes` The amount (in bytes) of memory to allocate.

DESCRIPTION

The `OTAllocMem` function allocates raw memory from a pool that Open Transport has created for a client application. This function returns a pointer to the allocated memory that the `OTFreeMem` function uses when to deallocate this memory.

SPECIAL CONSIDERATIONS

Do not make this call without having already called the `OTEnterInterrupt` function.

SEE ALSO

To free the memory you allocated with this function, use the `OTFreeMem` function (described next).

OTFreeMem

Frees memory allocated with the `OTAllocMem` function.

C INTERFACE

```
void OTFreeMem(void* arg)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`arg`	A pointer to the memory allocated by the `OTAllocMem` function. This pointer is returned by the `OTAllocMem` function.

SPECIAL CONSIDERATIONS

Do not make this call without having already called the `OTEnterInterrupt` function.

SEE ALSO

To allocate memory from the Open Transport pool, use the `OTAllocMem` function (page 7-21).

Idling and Delaying Processing

Open Transport provides idle and delay processing functions.

OTIdle

Idles your computer.

C INTERFACE

```
void OTIdle(void)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

DESCRIPTION

You can call the `OTIdle` function while you are waiting for asynchronous provider operations to complete. It is not necessary for the correct operation of Open Transport to call this function, but it provides compatibility for existing programs that use an idling function.

SPECIAL CONSIDERATIONS

You cannot call the `OTIdle` function at primary interrupt time. This function does not call the `SystemTask`, `WaitNextEvent`, or `GetNextEvent` functions.

OTDelay

Delays processing for a specified number of seconds. This function is only provided for compatibility with the UNIX `sleep` function.

C INTERFACE

```
void OTDelay(UInt32 seconds)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`seconds` The number of seconds to delay.

DESCRIPTION

The `OTDelay` function delays processing for the number of seconds specified in the `seconds` parameter. While the delay is occurring, `OTDelay` continuously calls the `OTIdle` function.

You can only call the `OTDelay` function from within an application at system task time. This function is only provided for compatibility with the UNIX `sleep` function to assist with portability of UNIX code.

SPECIAL CONSIDERATIONS

You should never call the `OTDelay` function in production code on a Macintosh computer.

Application-Defined Functions

To use either the system or the deferred task functions, you need to write a process callback function that Open Transport can call to execute your task at the appropriate time. When you create a system or a deferred task, you provide Open Transport with a pointer to your process callback function.

MyProcessCallbackFunction

Open Transport calls your process callback function when the scheduled task associated with it occurs. You use this function to specify Open Transport functions that you want to execute at specific system or deferred task times.

C INTERFACE

```
void MyProcessCallbackFunction(void* arg)
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`arg`	A pointer to application-defined data, typically providing context information.

DESCRIPTION

The `OTCreateSystemTask` and the `OTCreateDeferredTask` functions return a reference that identifies the newly created task so that you can schedule it for execution at specified times. Among other data, the task contains a pointer to your process callback function. When the scheduled task-execution time occurs, Open Transport calls your process callback function and passes it the data in the `arg` parameter.

SEE ALSO

For more information about how to use your process callback function, see the section "Using System and Deferred Tasks," beginning on page 7-4.

TCP/IP Services

Contents

This chapter describes the Open Transport implementation of TCP/IP, including the use of Open Transport endpoint and mapper functions with TCP/IP. This chapter also describes the TCP/IP service provider, which provides an interface to the TCP/IP Domain Name Resolver (DNR) for clients of Open Transport. To get the most out of this chapter, you should already be familiar with the concepts and application interfaces described in the chapters "Introduction to Open Transport," "Provider Manager," "Endpoints," "Mappers," "Option Management," and "Configuration Management" in this book. In addition, this chapter gives only a very rudimentary introduction to the TCP/IP protocol family. You will need to familiarize yourself with the operation and use of the various TCP/IP protocols before you can make effective use of the Open Transport implementation of TCP/IP. The following section, "About the TCP/IP Protocol Family," introduces the protocol family and gives pointers to more information.

This chapter describes TCP/IP-specific information about Open Transport functions and gives possible values for options that you can use with the TCP/IP protocols. You need this information only if you have a specific need to use the TCP/IP protocols or must bind explicitly to an IP address. If you are using Open Transport in a protocol-transparent fashion, you do not need the information in this chapter.

In this chapter the term *TCP/IP* is used when the information presented applies equally to all protocols of the TCP/IP family (such as RARP, BOOTP, DHCP, or UDP, as well as TCP and IP). When the information is specific to one protocol, the name of that protocol is used.

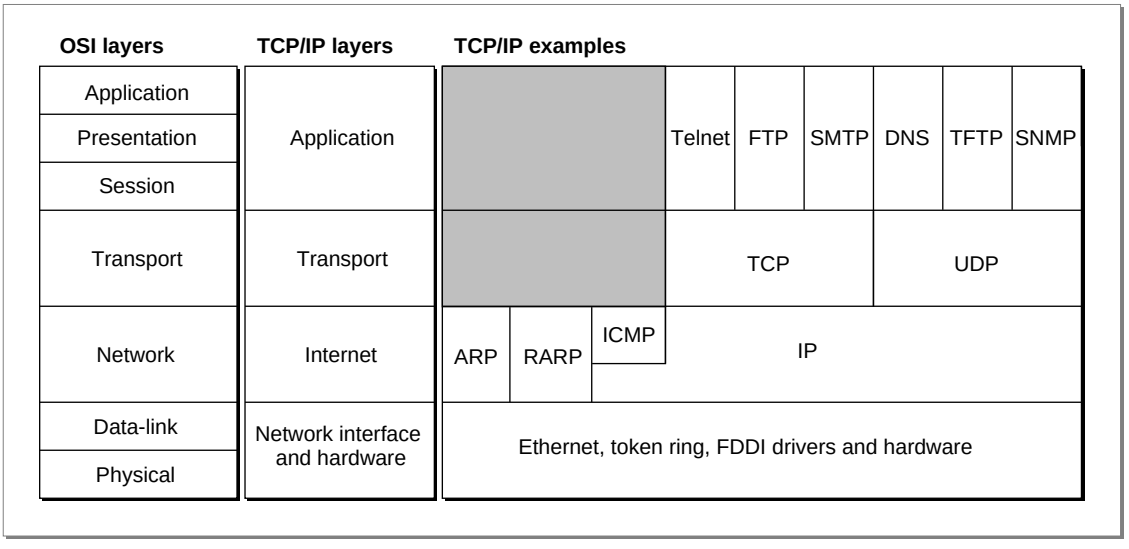
This chapter starts with a brief introduction to the TCP/IP protocol family, followed by an introduction to the TCP/IP services provided by Open Transport. The section "Using General Open Transport Functions With TCP/IP," beginning on page 8-14 describes TCP/IP-specific information relating to functions described in the chapters "Endpoints" and "Mappers" in this book. The reference section describes those data structures and functions available only to users of the Open Transport TCP/IP implementation. In addition, "Options," beginning on page 8-29 describes the TCP/IP options you can specify when you configure a provider.

About the TCP/IP Protocol Family

The **TCP/IP protocol family** is a set of networking protocols in wide use throughout the world for government and business applications. The TCP/IP protocol family includes a basic datagram-delivery protocol, called **Internet Protocol (IP);** a connectionless datagram protocol called **User Datagram Protocol (UDP)** that segments data to handle larger datagrams than those allowed by IP; and a connection-oriented data stream protocol that provides highly reliable data delivery, called **Transmission Control Protocol (TCP).** In addition to these three fundamental protocols, TCP/IP includes a wide variety of protocols for specific uses, mostly at the application-protocol level.

Figure 8-1 shows the TCP/IP functional layers and examples of TCP/IP protocols that run in each layer. For purposes of comparison, Figure 8-1 also shows the OSI model functional layers. Note that reliability of data delivery can depend on the reliability built into TCP, or can be added at the application level by protocols using UDP. Similarly, a protocol based on UDP can implement connection-oriented services at the application-protocol level.

Figure 8-1 TCP/IP protocols and functional layers



As discussed in the chapter "Endpoints" in this book, the way you use Open Transport functions to send data depends both on whether the protocol you wish to use is connection-oriented and whether it is transaction-based. Table 8-1 shows how the TCP/IP protocols provided with Open Transport fit into this matrix. Notice that the TCP/IP protocol family offers no transaction-based protocols at the transport or internet layers of the TCP/IP architecture.

Open Transport provides an application interface to the IP protocol known as **RawIP**, as shown in Table 8-1. For more information on this interface to the IP protocol, see "Using RawIP" on page 8-11.

Table 8-1 The Open Transport protocol matrix and TCP/IP protocols

	Connectionless	**Connection-oriented**
Transactionless	RawIP UDP	TCP
Transaction-based		

Open Transport offers interfaces to the TCP, UDP, and IP protocols, and to the domain name resolver (DNR). Only those protocols are discussed in the rest of this chapter. Open Transport also provides implementations of the RARP, BOOTP, and DHCP protocols, but those protocols are used by Open Transport for automatic configuration of a host, and they have no application interfaces.

For general information about the other protocols shown in Figure 8-1, see any good book on TCP/IP. Two such books for information on TCP/IP protocol internals are *TCP/IP Illustrated, Volume 1* by W. Richard Stevens and *Internetworking with TCP/IP, Volume 1* by Douglas E. Comer.

The Open Transport TCP/IP software modules are based on the UNIX Streams architecture. For more information about Streams, see *UNIX System V Release 4: Programmer's Guide: STREAMS*.

The Open Transport API is based on the XTI standard as documented in *X/Open CAE Specification (1992): X/Open Transport Interface (XTI)*.

The TCP/IP protocols are defined in a series of documents called Requests for Comments (RFCs). RFCs are available over the Worldwide Internet or from the Defense Data Network (DDN) Network Information Center (NIC) at

DDN Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

Telephone: 800-365-3642

You can get information on how to obtain RFCs via e-mail by sending an e-mail message to "rfc-info@ISI.EDU". The message body must read "help: ways_to_get_rfcs".

In addition, any program that implements the file transfer protocol (FTP) can download copies of the RFC list and the RFCs themselves from the internet address "nic.ddn.mil."

About TCP/IP Services

The TCP/IP services provided by Open Transport include implementations of the TCP, UDP, RARP, BOOTP, DHCP, and IP protocols, an application interface to the domain name resolver (DNR), and utility functions you can use when creating and resolving internet addresses. You can open TCP, UDP, and RawIP endpoints and DNR mappers using the interfaces described in the chapters "Endpoints" and "Mappers" in this book .

A **domain name resolver** translates between the character-string names used by people to identify nodes on the internet and the 32-bit internet addresses used by the network itself. In that sense, its function is similar to AppleTalk's Name-Binding Protocol (NBP). Unlike AppleTalk, however, TCP/IP protocols do not specify a way for clients to register a name on the network. Instead, the network administrator must maintain a server that stores the character-string names and internet addresses of the servers on the internet, or each individual host must keep a file of such names and addresses. The Open Transport implementation of TCP/IP includes a DNS stub name resolver; that is, a software module that can use the services of the domain name system (DNS) to resolve a name to an address.

The nodes on a TCP/IP internet are known as **hosts**. A host that is addressable by other hosts has a host name and one or more domain names that identify the hierarchically arranged **domains,** or collections of hosts, to which it belongs. For example, the Open Transport team, part of the system software group at Apple Computer, might have a server with a fully qualified domain name of "otteam.ssw.apple.com." In this case, "otteam" represents the domain of hosts belonging to the Open Transport team, "ssw" represents the domain of hosts belonging to the system software group (which includes the Open Transport team plus several other teams), and so forth. A **fully qualified domain name** corresponds to an **internet address,** also known as an **IP address,** which is a 32-bit number that uniquely identifies a host on a TCP/IP network. An internet address is commonly expressed in dotted-decimal notation (for example, "12.13.14.15") or hexadecimal notation (for example, "0x0c0d0e0f").

To use the application interface to Open Transport's DNR, you must first open a TCP/IP service provider. Once you have done so, you can

- resolve a domain name to one or more associated internet addresses
- look up the domain name associated with an internet address
- retrieve the character strings stored by the domain name server that describe a host's processor and operating system
- retrieve DNS information associated with any query class and type
- obtain a list of mail exchanges and mail preference values for a host to which you wish to deliver mail

A **mail exchange** is any host that can accept mail for another host or for a domain. A mail exchange can be a mail server, a router, or just a host configured to accept and pass on mail. A **mail preference value** is used by a mail application to determine to which mail exchange to deliver a message when there is more than one that can accept mail for a particular domain. The mailer sends the mail to the mail exchange with the lowest preference value first and tries the others in turn until the mail is delivered or until the mailer deems the mail undeliverable.

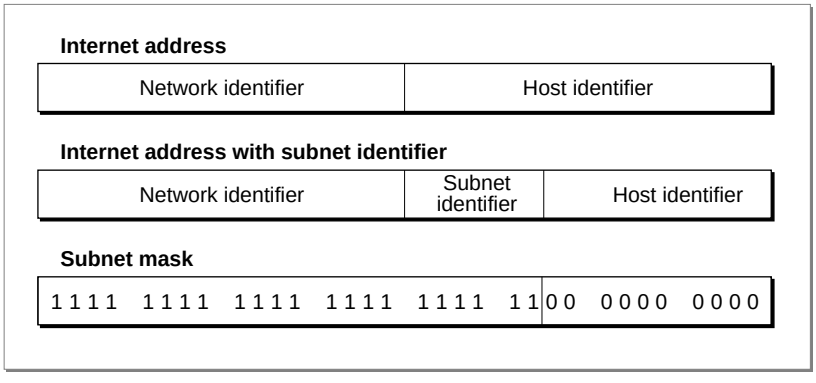
The Open Transport TCP/IP services also include several utility functions. You can use these functions to

- get internet addresses and subnet masks for all the TCP/IP interfaces on the local host
- fill in data structures used for internet addresses

- convert an IP address string from dotted-decimal notation or hexadecimal notation to a 32-bit IP address
- convert a 32-bit IP address into a character string in dotted-decimal notation

The **subnet mask** determines what portion of the IP address is dedicated to the host identifier and what portion identifies the subnet. A **subnet** is a portion of a network, which is in turn a portion of an internet. Figure 8-2 illustrates the subnet portion of an address. The top portion of the figure shows an internet address that does not include a subnet identifier. The center portion of the figure shows an internet address that includes a subnet. Notice that the subnet identifier is formed by using a portion of the bits reserved for the host identifier. The bottom portion of the figure shows the subnet mask, which you can use to determine how many bits are used for the subnet and how many are used for the host.

Figure 8-2 Internet subnet address



Note
As used in this chapter, a *TCP/IP interface* is the point of attachment of a host to a TCP/IP network. In the case of a multihomed host, the user can configure more than one TCP/IP interface. At present, the architecture of Open Transport TCP/IP supports multihoming, but it is not yet possible to configure a multihomed host. Therefore, all functions designed to return information about all the TCP/IP interfaces on a host return information about a single interface. ◆

About the Open Transport DNR

The functions described in "Resolving Internet Addresses," beginning on page 8-42 and "Getting Information About an Internet Host," beginning on page 8-45 are implemented by the Open Transport domain name resolver (DNR). The DNR also implements the `OTLookupName` function (page 8-20) when you configure a TCP/IP mapper. The DNR can be invoked by a UDP endpoint's call to the `OTSndUData` function, a TCP endpoint's call to the `OTConnect` function, or a call to the `OTResolveAddress` function by either type of endpoint. This section describes how the Open Transport DNR operates.

When Open Transport loads TCP/IP, Open Transport initializes the DNR with a prioritized list of search domains, a prioritized list of name servers to be searched, and, if the user has a Hosts file, a list of name-to-address mappings and a list of canonical name-to-alias mappings from the Hosts file. Hosts files are described in "Using the Hosts File" on page 8-12.

When you request that the DNR resolve a name to an address, the DNR checks for a period (.) at the end of the name. If it finds one, the DNR assumes the name to be a fully qualified domain name and attempts to resolve it. If the name contains no periods, the DNR assumes that the name is partially qualified and begins a search in the first of the configured search domains. If the name contains at least one period, but doesn't have one at the end of the name, the DNR first assumes the name is fully qualified and attempts to resolve it as such. If that process fails, then the DNR assumes that the name is partially qualified and starts a search as for any other partially qualified name.

For each search domain, the DNR calls the configured name servers in the order specified during configuration. The DNR returns the first answer it finds and terminates the search. If there is no other way to resolve the name or if the DNR is looking for the name that corresponds to an address, it searches the root domain. The DNR abandons its search if it hasn't found the name in a predetermined amount of time.

The Open Transport DNR implements only the following DNS query types.

Type	Description
A	Resolve name to 32-bit IP host address.
HINFO	Return type of processor (CPU) and operating system of host.
MX	Return name of mail exchange for the domain.
PTR	Resolve address to a fully qualified domain name.

The Open Transport DNR does not cache negative responses or partial name resolutions from name servers; it depends on full-service name servers to fully resolve the name and then caches only the final resolved name.

The DNR always requests recursion (that is, it requests the name server it contacts to contact other name servers, as required, to completely resolve a name). However, if the name server does not perform recursion but does provide references to additional name servers, the DNR follows up on these references. The DNR does not save the name server references after the name resolution is complete; instead, it starts over each time with the configured name servers.

The DNR caches name-to-address and canonical name-to-alias mappings, but not host information (CPU and operating-system types) or the results of mail exchange (MX) queries.

Open Transport supports use of a Hosts file, which the DNR uses to supplement and customize its cache. The Hosts file is located in the Preferences folder in the System Folder. However, it is best to avoid using a Hosts file if a name server is available, as Hosts files usually waste memory in the local cache and degrade performance of the DNR. If you do use a Hosts file, keep it as short as possible. For more information about the Hosts file, see "Using the Hosts File" on page 8-12.

Using TCP/IP Services

This section describes how to use the Open Transport RawIP interface, how to implement IP multicasting, and how to use a variety of Open Transport endpoint and mapper functions with the TCP/IP protocols. TCP/IP options are described in "Options," beginning on page 8-29.

Using RawIP

The Open Transport TCP/IP software modules provide a RawIP interface to the IP protocol. The RawIP interface is provided to facilitate the implementation of new protocols that use IP for datagram delivery. Therefore, in order to use a RawIP endpoint, you must specify a value for the Protocol field in the IP datgram header. By default, RawIP fills in the Protocol field with a 1, indicating the packet carries an ICMP message. You can specify a different protocol type for a RawIP endpoint by using the option `XTI_PROTOTYPE`, described in the chapter "Option Management" in this book. The option value consists of a longword containing the number of the protocol to be used by the RawIP endpoint.

All RawIP endpoints using a specific protocol receive a copy of any inbound packets destined for that protocol. Thus, if several programs were using ICMP on the same host, they would all receive a copy of every inbound ICMP datagram.

The data delivered to a RawIP endpoint includes the full IP header, which is 20 bytes long if it includes no options.

▲ **WARNING**
If you open a RawIP endpoint, you are responsible for implementing the protocol that is a client of IP running over that endpoint. Because an improperly implemented protocol can cause the host to crash or cause the loss of data on the network, you should not attempt to use RawIP unless you are an expert network programmer. ▲

Using IP Multicasting

Open Transport TCP/IP provides IP multicasting level 2, as described in RFC 1112. To join a multicast group, use the `IP_ADD_MEMBERSHIP` option (page 8-37), passing in a `TIPAddMulticast` structure to specify the address and network interface of the group you wish to join. For a multihomed system, you can use the value `kOTAnyInetAddress` for the interface address to use the default multicast interface.

The time-to-live value for outbound multicast data defaults to 1; you can use the `IP_MULTICAST_TTL` option to set a different value. The time-to-live value is a hop count: Each router that processes the datagram decrements the Time to Live field and discards the datagram if the value reaches 0. Because every router that receives a multicast packet forwards it, a high time-to-live value for a multicast packet can cause the packet to propagate widely through the internet. Therefore, keep this value as low as possible.

By default, Open Transport IP loops back multicast datagrams to any member of the group on the sending machine. Pass a value of `T_NO` to the `IP_MULTICAST_LOOP` option to turn off loopbacks.

Using the Hosts File

The Open Transport DNR supports use of a Hosts file, which is located in the Preferences folder in the System Folder. The DNR parses the Hosts file when TCP/IP is loaded and uses it to initialize its list of host addresses, aliases, and name servers.

Note
Because the DNR downloads the information in the Hosts file and keeps it in RAM, a large Hosts files waste memory and degrades the performance of the DNR. Therefore, you should avoid using a Hosts file, or if you must use one, keep the Hosts file as small as possible. ◆

The Hosts file can include lines of data, blank lines, and comments. A comment begins with a semicolon (;) and can be on the same line as data. The comment extends from the semicolon to the end of the line.

A data line in the Hosts file has either of the following two formats (the square brackets indicate an optional element):

domain-name [*time-to-live* `IN`] *type rdata*

domain-name [`IN` *time-to-live*] *type rdata*

The elements are defined as follows:

domain-name	An absolute or fully qualified domain name unless the data type is `CNAME`, in which case the name can be an alias. It need not be terminated by a period (.), but must contain at least one period internally.
time-to-live	The record's configured lifetime, in seconds. If this field is not present, the DNR assumes the entry has an infinite lifetime. You can also specify –1 for the time to live to indicate an infinite lifetime.
`IN`	The data class; always `IN` indicating the internet domain.
type	The type of data. This field can be `A` for a host address, `CNAME` for the canonical name of an alias, or `NS` for a name server. Other DNS types are not supported in the Hosts file.
rdata	The data. For the `A` data type, this field must be an address in dotted-decimal format. For the `CNAME` and `NS` data types, this field must be a fully qualified domain name.

Here are some sample data lines:

```
apple.com A 130.43.2.2 ;address of host apple.com

rivers IN CNAME rivers.apple.com        ;canonical name of the host
                                        ; whose local alias is "rivers"

mit.edu. 86400 NS achilles.mit.edu      ;name server for mit.edu
                                        ; domain. Entry has a 1-week
                                        ; lifetime
```

Querying DNS Servers

In addition to the explicit simplified functions that are provided for the most commonly made queries such as name-to-address, address-to-name, system

CPU and OS, and mail exchange queries, there is a generic query function, `OTInetQuery`, that you can use for any DNS query.

The `OTInetQuery` function allows you to use the Domain Name Resolver (DNR) for generic domain name service (DNS) queries. You can ask for any query type and class, and in response, Open Transport returns as many `DNSQueryInfo` structures as it can fit in the buffer you provide.

There are three types of responses: answers, authority responses, or additional information, and there are typically several of each type. Each response has its own `DNSQueryInfo` structure, with all the answers first, then all the authority records, then all the additional information. Authority responses refer you to DNS servers and other sources that may have helpful information for this answer and additional responses provide address data for the servers and sources referred to in the authority records.

If, for example, you use the `OTInetQuery` function to find out the IP addresses for a name, you might get back 13 `DNSQueryInfo` structures in your answer buffer: With, say, 2 IP address structures, 4 authority responses, and 7 additional information responses.

To help you parse this huge answer buffer, Open Transport provides two optional parameters for the `OTInetQuery` function, `argv` and `argvlen`, that create an array pointing to the individual responses.

Using General Open Transport Functions With TCP/IP

This section describes any special considerations that you must take into account for Open Transport functions when you use them with the Open Transport TCP/IP implementation. You should be familiar with the function descriptions in the chapters “Endpoints” and “Mappers” in this book before reading this section.

Obtaining Endpoint Data With TCP/IP

The following values can be returned by the `info` parameter to the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, and `OTGetEndpointInfo` functions when used with TCP/IP protocols:

Parameter	TCP	RawIP and UDP	Meaning
`info->addr`	`T_INVALID`	`T_INVALID`	No access to transport protocol addresses is provided.
	Greater than 0	Greater than 0	Maximum size of a protocol address.
`info->options`	`T_INVALID`	`T_INVALID`	No options.
	Greater than 0	Greater than 0	Maximum number of bytes needed to hold protocol-specific options.
`info->tsdu`	0		TSDUs not supported.
		`T_INVALID`	Transfer of normal data not supported.
		Greater than 0	Maximum size of a TSDU.
`info->etsdu`	`T_INFINITE`		No limit on the size of an ETSDU.
		`T_INVALID`	Transfer of expedited data not supported.
`info->connect`	`T_INVALID`	`T_INVALID`	Data cannot be sent with functions that establish connections.
`info->discon`	`T_INVALID`	`T_INVALID`	Data cannot be sent with abortive disconnects.
`info->servtype`	`T_COTS`		Connection mode supported; orderly disconnects not supported.
	`T_COTS_ORD`		Connection mode and orderly disconnects supported.
		`T_CLTS`	Connectionless mode supported.
`info->flags`	`T_SENDZERO`	`T_SENDZERO`	Zero-length TSDUs supported.

IMPORTANT

The preceding table shows only what values are possible for each protocol. Be sure to to use the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, or `OTGetEndpointInfo` function to obtain the current values for these parameters. ▲

These fields and the significance of their values are described in more detail in the document *X/Open CAE Specification: X/Open Transport Interface (XTI)*.

Using Endpoint Functions With TCP/IP

This section describes protocol-specific information about functions described in the chapter "Endpoints" in this book. The functions are listed in the same order that they appear in that chapter.

OTBind

The `OTBind` function associates a local protocol address with the endpoint you specify. You must use this function for every protocol.

The `addr` field of the `TBind` structure refers to the local endpoint and so must specifically include a port number. Use an `InetAddress` structure (described in "Internet Address Structure" on page 8-23) to specify this address.

Because the architecture of Open Transport TCP/IP provides for multihoming (although this feature has not yet been implemented), you can specify an IP address of `kOTAnyInetAddress` to indicate that your application or process will accept packets from any TCP/IP interface that the user has configured in the TCP/IP control panel.

If you specify an address of 0 to the `OTBind` function, then the `OTGetProtAddress` function always returns an IP address of 0. In that case, you must use the `OTInetGetInterfaceInfo` function (page 8-52) to determine the IP address of a running IP interface. However, if you pass in a valid address with a port number of 0, the TCP/IP service provider assigns a port for you and the `OTGetProtAddress` function returns the assigned port number.

You can use the `OTInetGetInterfaceInfo` function (page 8-52) to get the IP addresses of all currently configured IP interfaces. Then, if you wish to receive packets from only a single interface, you can bind the endpoint to the address for that interface.

OTLook

The `OTLook` function checks for asynchronous events such as incoming data or connecton requests. You can use this function with any protocol.

As soon as a segment with the TCP urgent pointer set (that is, expedited data) enters the TCP receive buffer, TCP posts the `T_EXDATA` event. The `T_EXDATA` event

remains posted until you have retrieved all data up to the byte pointed to by the TCP urgent pointer.

OTGetProtAddress

If you bind an endpoint to an IP address of 0 in order to accept packets from any valid TCP/IP interface, then the `OTGetProtAddress` function always returns an IP address of 0. This is because in a multihomed machine, there is a separate IP address for each interface, and there's no way for open transport to know which one you want. in that case, you must use the `OTInetGetInterfaceInfo` function (page 8-52) to determine the IP address of a running IP interface. On the other hand, if you bind an endpoint to a specific interface, the `OTGetProtAddress` function returns the address of that interface, as expected.

OTConnect

The `OTConnect` function requests a connection to a specified remote endpoint. You can use this function with TCP but not with UDP or IP.

The `rcvcall->addr` field returns the same `TNetbuf` structure you specify in the `sndcall->addr` field.

Because TCP does not allow you to send any application-specific data during the connection establishment phase, you must set the `sndcall->udata.len` field to 0. TCP ignores any data in the `sndcall->udata.buf` field.

Note that TCP, not the receiving application, confirms the connection.

OTRcvConnect

The `OTRcvConnect` function reads the status of a previously issued connection request. You can use this function with TCP, but not with UDP or IP.

Because TCP does not allow you to send any application-specific data during the connection establishment phase, you must set the `call->udata.maxlen` field to 0. TCP ignores any data in the `call->udata.buf` field.

On return, the `call->addr` field points to the fully qualified internet address of the endpoint that accepted the connection.

OTListen

The `OTListen` function listens for an incoming connection request. You can use this function with TCP, but not with UDP or IP.

When the `OTListen` function successfully completes execution (that is, when you receive the `T_LISTEN` event), the `call` parameter describes a connection that has already been completed at the TCP level. You use the `OTAccept` function to complete a connection at the application level. If you wish to reject a connection, you must call the `OTSndDisconnect` function after the `OTListen` function successfully completes execution.

Because TCP does not allow you to send any application-specific data during the connection establishment phase, you must set the `call->udata.len` field to 0. TCP ignores any data in the `call->udata.buf` field.

OTAccept

The `OTAccept` function accepts an incoming connection request. You can use this function with TCP, but not with UDP or IP.

Because TCP does not allow you to send any application-specific data during the connection establishment phase, you must set the `call->udata.len` field to 0. TCP ignores any data in the `call->udata.buf` field.

If you wish to send either of the association-related options (`IP_OPTIONS` or `IP_TOS`) with the connection confirmation, you must use the `OTOptionManagement` function to set the values of these options before you receive the `T_LISTEN` event. TCP has already established a connection when you receive the `T_LISTEN` event, and it is too late for the `OTAccept` function to negotiate these options.

OTSndUData

The `OTSndUData` function sends data through a connectionless transactionless endpoint. You can use this function with UDP and IP, but not with TCP.

The current value for the maximum size of a RawIP or UDP datagram is returned in the `info->tsdu` parameter of the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, and `OTGetEndpointInfo` functions.

OTSnd

The `OTSnd` function sends data through a connection-oriented transactionless endpoint. You can use this function with TCP, but not with UDP or IP.

TCP ignores the `OTSnd` function's `T_MORE` flag.

If you set the `T_EXPEDITED` flag, you must send at least 1 byte of data. If you call the `OTSnd` function with more than 1 byte specified and the `T_EXPEDITED` flag set, the TCP urgent pointer points to the last byte of the buffer.

OTRcv

The `OTRcv` function receives data through a connection-oriented endpoint. You can use this function with TCP, but not with UDP or IP.

Because TCP ignores the `T_MORE` flag when it is sending data and does not transmit the flag, you should ignore the `T_MORE` flag when receiving normal data. However, if a byte in the data stream is pointed to by the TCP urgent pointer, TCP receives this byte and as many bytes as possible preceding the marked byte with the `T_EXPEDITED` flag set. If your buffer is too small to receive all of the expedited data, TCP sets the `T_MORE` flag as well. Note that this situation might result in the number of bytes received as expedited data not being equal to the number of bytes sent by the originator as expedited data.

OTSndDisconnect

The `OTSndDisconnect` function initiates an abortive disconnect or rejects a connection request. You can use this function with TCP, but not with UDP or IP.

Because TCP does not allow you to send any application-specific data during a disconnect, you must set the `call->udata.len` field to 0. TCP ignores any data in the `call->udata.buf` field.

OTRcvDisconnect

The `OTRcvDisconnect` function returns information about why a connection attempt failed or an established connection was terminated. You can use this function with TCP, but not with UDP or IP.

Because TCP does not allow you to send any application-specific data during a disconnection, you must set the `discon->udata.len` field to 0. TCP ignores any data in the `discon->udata.buf` field.

Using Mapper Functions With TCP/IP

This section describes protocol-specific information about functions described in the chapter "Mappers" in this book. The functions are listed in the same order that they appear in that chapter.

OTRegisterName

Because the TCP/IP domain name system does not include a method for clients to register their names on the network, the Open Transport domain name resolver (DNR) does not support the `OTRegisterName` function. If you call this function for a TCP/IP mapper, it will return the `kOTNotSupportedErr` result code.

OTDeleteName

This function is not supported by the TCP/IP domain name resolver (DNR). If you call this function for a TCP/IP mapper, it will return the `kOTNotSupportedErr` result code.

OTLookupName

You can use the `OTLookupName` function to resolve a domain name to an internet address. Specify the name as a character string pointed to by the `request->udata.buf` parameter. The name can be just a host name (otteam), a partially qualified domain name ("otteam.ssw"), a fully qualified domain name ("otteam.ssw.apple.com."), or an internet address in dotted-decimal format ("17.202.99.99"), and can optionally include the port number ("otteam.ssw.apple.com:25" or "17.202.99.99:25").

The function returns a pointer to the address in the `reply->udata.buf` parameter. The address is in the format of an `InetAddress` structure (page 8-23), which includes the address type, the port number, and the IP address in hexadecimal format. If you don't specify a port number, the returned `InetAddress` structure contains a port number of 0. You can use this address directly in all Open Transport functions that require an internet address, such as `OTConnect`, `OTSndUData`, and `OTBind`.

The `OTLookupName` function returns only a single address, regardless of how many addresses are known for a single multihomed host. To obtain a list of up to 10 addresses for a multihomed host, use the `OTStringToAddress` function (page 8-42).

TCP/IP Services Reference

This section describes the data structures and functions provided by the TCP/IP service provider.

Constants and Data Types

This section describes the data types used by the TCP/IP service interface.

Basic Types and Constants

The data types defined in this section are used by other structures and functions of the Open Transport API. They are not specific to TCP/IP, but are included here for your convenience.

```
typedef unsigned char UInt8;
typedef unsigned short UInt16;
typedef unsigned long UInt32;

typedef char SInt8;
typedef short SInt16;
typedef long SInt32;

typedef UInt16 InetPort;
typedef UInt32 InetHost;
```

You can use the protocol names `kTCPName`, `kUDPName`, and `kRawIPName` when calling the `OTCreateConfiguration` function to configure an endpoint. You can use the protocol name `kDNRName` when calling the `OTCreateConfiguration` function to configure a mapper. The `OTCreateConfiguration` function is described in the chapter "Configuration Management" in this book.

```
#define kDNRName                'dnr'
#define kTCPName                'tcp'
#define kUDPName                'udp'
#define kRawIPName              'rawip'
```

You can use the constant `kDefaultInternetServicesPath` to create a TCP/IP service provider. Its value is a pointer to a configuration structure, so you do not use the `OTCreateConfiguration` function with this constant. Instead you use this constant as a parameter when calling the `OTAsyncOpenInternetServices` and the `OTOpenInternetServices` functions that create TCP/IP service providers.

```
#define kDefaultInternetServicesPath((OTConfiguration*)-3)
```

You use the `AF_INET` and `AF_DNS` values as address types when filling in address structures (see "Internet Address Structure" on page 8-23 and "DNS Address Structure" on page 8-24). You can use the `kOTAnyInetAddress` value with the `OTBind` function (page 8-16).

```
enum {
        AF_INET             = 2,,       /* TCP/IP address */
        AF_DNS              = 42        /* domain name server address */
};
```

You can use the `kOTAnyInetAddress` value with the `OTBind` function (page 8-16).

```
enum {
        kOTAnyInetAddress       = 0     /* wildcard */
};

enum {
        kMaxHostAddrs           = 10,
        kMaxSysStringLen        = 32,
        kMaxHostNameLen         = 255
};
typedef char InetDomainName[kMaxHostNameLen];
```

Note
The maximum valid domain-name length for fully qualified domain names includes the trailing period (.). Names not terminated with a period are limited to 254 bytes. ◆

The following completion event codes are returned by TCP/IP service provider functions:

```
enum {
        T_DNRSTRINGTOADDRCOMPLETE           = kPRIVATEEVENT+1,
        T_DNRADDRTONAMECOMPLETE             = kPRIVATEEVENT+2,
        T_DNRSYSINFOCOMPLETE                = kPRIVATEEVENT+3,
        T_DNRMAILEXCHANGECOMPLETE           = kPRIVATEEVENT+4,
        T_DNRQUERYCOMPLETE                  = kPRIVATEEVENT+5
};

enum {
        kDefaultInetInterface               = -1,
        kInetInterfaceInfoVersion           = 2
};

enum {
        kPRIVATEEVENT   = (OTEventCode)0x10000000
};
```

The following are miscellaneous TCP/IP-related constants. Some, such as the version number, are subject to change.

```
#define kInetVersion "3.0"      /* MacTCP 3.0*/

#define kInetPrefix "ot:inet$"

#define SET_TOS(prec,tos)       (((0x7 & (prec)) << 5) | (0x1c & (tos)))
```

Internet Address Structure

You use the internet address structure when providing a TCP or UDP address to the Open Transport functions `OTConnect`, `OTSndURequest`, and `OTBind`. You can use the `OTInitInetAddress` function (page 8-53) to fill in an internet address structure. The internet address structure is defined by the `InetAddress` data type.

```
struct InetAddress {
        OTAddressType       fAddressType;   /* address type; always AF_INET */
        InetPort            fPort;          /* port number */
```

```
        InetHost            fHost;          /* host address (network byte order) */
        UInt8               fUnused[8];     /* reserved */
};
```

Field descriptions

`fAddressType`	The address type. The only possible value for this field is `AF_INET`, which identifies the TCP/IP protocol family.
`fPort`	The port number.
`fHost`	The IP address of the host in network byte order (that is, high-order byte first) in hexadecimal notation.
`fUnused`	Reserved.

DNS Address Structure

You can use the DNS (domain name system) address structure with the `OTConnect` function when you are creating a TCP endpoint, with the `OTSndUData` function when you are creating a UDP endpoint, or with the `OTResolveAddress` function with either TCP or UDP endpoints. If you do so, the domain name resolver (DNR) will resolve the address for you automatically. You can use the `OTInitDNSAddress` function (page 8-55) to fill in a DNS address structure. The DNS address structure is defined by the `DNSAddress` data type.

```
struct DNSAddress {
    OTAddressType       fAddressType;   /* address type; always AF_DNS */
    InetDomainName      fName;          /* domain name */
};
```

Field descriptions

`fAddressType`	The address type. The only possible value for this field is `AF_DNS`.
`fName`	The address to be resolved by the DNR.

This address you specify can be just the host name (“otteam”), a partially qualified domain name (“otteam.ssw”), a fully qualified domain name (“otteam.ssw.apple.com.”), or an internet address in dotted-decimal format (“17.202.99.99”), and can optionally include the port number (“otteam.ssw.apple.com:25” or “17.202.99.99:25”).

Because the port number is not actually part of the domain name, it is possible to have a domain name–port number combination that exceeds 255 bytes. If

you wish to specify such a string, you must provide a structure based on the DNS address structure that has sufficient space to contain the full string. In any case, the domain name itself cannot exceed 255 bytes.

You can use the DNS address structure to provide an unresolved TCP address with the `OTConnect` function or a UDP address with the `OTSndUData` function. To do so, specify a pointer to the DNS address structure as the `udata->addr.buf` parameter in your call to the `OTSndUData` function or as the `sndCall->addr.buf` parameter in your call to the `OTConnect` function.

DNS Query Information Structure

The DNS query information structure is used by the Domain Name Resolver (DNR) to return answers to DNS queries made using the `OTInetQuery` function. The DNS query information structure is defined by the `DNSQueryInfo` data type.

```
struct DNSQueryInfo {
    UInt16          qType;
    UInt16          qClass;
    UInt32          ttl;
    InetDomainName  name;
    UInt16          responseType;
    UInt16          resourceLen;
    char            resourceData[4];
};
typedef struct DNSQueryInfo DNSQueryInfo;
```

Field descriptions

`qType` The DNS query type, such as `MX` and `PTR`, for which you wish to query.

`qClass` The DNS query class, such as `Inet` and `Hesiod`, for which you wish to query.

`ttl` An integer indicating the resource record's time to live (in seconds). To reduce network overhead, keep this value as low as possible.

`name` The fully qualified domain name or address for which you are making the query. In the case of a `CNAME` data types, the name can be an alias or a partially qualified domain name, and if you use either of these, the `OTInetQuery` function returns a fully qualified canonical name.

`responseType`	The type of response. This can be an answer (a value of 2), an authority response (a value of 3), or additional information (a value of 4). Answers provide the information you are seeking (such as a resolved internet address), authority responses refer you to DNS servers and other sources that may have helpful information for this answer, and additional responses provide address data for the servers and sources referred to in the authority responses.
`resourceLen`	The actual length of the resource data returned.
`resourceData`	The resource data that is returned. This is at least 4 bytes long, and is usually longer.

Internet Interface Information Structure

The `OTInetGetInterfaceInfo` function (page 8-52) returns information about the local host in an internet interface information structure. The internet interface information structure is defined by the `InetInterfaceInfo` data type.

```
struct InetInterfaceInfo {
    InetHost            fAddress;               /* host address */
    InetHost            fNetmask;               /* subnet mask */
    InetHost            fBroadcastAddr;         /* broadcast address */
    InetHost            fDefaultGatewayAddr;    /* default gateway
                                                       address */
    InetHost            fDNSAddr                /* DNS address*/
    UInt16              fVersion;               /* version number */
    UInt16              fHWAddrLen              /* HW addr length */
    UInt8*              fHWAddr;                /* HW address */
    UInt32              fIfMTU                  /* Max Trans Unit */
    UInt8*              fReservedPtrs[2];       /* reserved */
    InetDomainName      fDomainName;            /* host's domain name */
    UInt8               fReserved[256];         /* reserved */
};
```

Field descriptions

`fAddress`	The IP address of the local host in network byte order (that is, high-order byte first) in hexadecimal notation.
`fNetMask`	The subnet mask of the local IP network.

`fBroadcastAddr`	The broadcast address for the interface. The broadcast address can also be calculated by performing an `OR` operation on the address with the complement of the subnet mask.
`fDefaultGatewayAddr`	The IP address of the default gateway. The default gateway is a router or gateway that can forward any packet destined outside the locally connected subnet. This information is optional; if you don't have a default gateway, you need a specific route for every packet or message sent to a subnet other than your own your directly connected subnet.
`fDNSAddr`	The address of a domain name server. This value can be returned by a server or typed in by the user during configuration of the TCP/IP interface.
`fVersion`	The version of the `OTInetGetInterfaceInfo` function; currently equal to 1.
`fHWAddrLen`	The length (in bytes) of the hardware address.
`fHWAddr`	A pointer to the hardware address.
`fIfMTU`	The maximum transmission unit size permitted for this interface's hardware.
`fReservedPtrs`	Reserved.
`fDomainName`	The default domain name of the host. This name doesn't include the host name.
`fReserved`	Reserved.

Internet Host Information Structure

The `OTInetStringToAddress` function (page 8-42) returns IP addresses for a host in an internet host information structure. The internet host information structure is defined by the `InetHostInfo` data type.

```
struct InetHostInfo {
    InetDomainName      name;
    InetHost            addrs[kMaxHostAddrs];
};
```

Field descriptions

`name` The **canonical name** of the host. The canonical name is a fully qualified domain name that is not an alias.

`addrs` Up to ten IP addresses associated with this host name. Only multihomed hosts have more than one IP address.

Internet System Information Structure

The `OTInetSysInfo` function (page 8-46) returns information about a host in an internet system information structure. The internet system information structure is defined by the `InetSysInfo` data type.

```
struct InetSysInfo {
    char            cpuType[kMaxSysStringLen];
    char            osType[kMaxSysStringLen];
};
```

Field descriptions

`cpuType` The CPU type of the specified host. This is a displayable character string maintained by the domain name server.

`osType` The operating system running on the specified host. This is a displayable character string maintained by the domain name server.

IP Multicast Address Structure

You use the IP multicast address structure with the `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` options (page 8-37) when you are adding or dropping membership in an IP multicast address. The IP multicast address structure is defined by the `TIPAddMulticast` data type.

```
struct TIPAddMulticast {
    InetHost multicastGroupAddress;
    InetHost interfaceAddress;
};
```

Field descriptions

`multicastGroupAddress`
: The IP address of the multicast group for which you want to add or drop membership.

`interfaceAddress`
: The IP address of the network interface that you are using for the multicast group.

Internet Mail Exchange Structure

The `OTInetMailExchange` function (page 8-47) returns host names and mail preference values in an array of internet mail exchange structures. The internet mail exchange structure is defined by the `InetMailExchange` data type.

```
struct InetMailExchange {
    UInt16              preference;
    InetDomainName      exchange;
};
```

Field descriptions

`preference`
: The mail exchange preference value. The mail exchanger with the lowest preference number is the first one to which mail should be sent.

`exchange`
: The fully qualified domain name of a host that can accept mail for your target host.

Options

This section describes the TCP, UDP, and IP options that you can use with provider functions such as `OTOptionManagement`, `OTConnect`, `OTSndUData`, or `OTSndURequest`.

Protocol Levels

The protocol level specifies the protocol to which the option applies. You specify the protocol level in the `level` field of the `TOption` structure when you specify an option.

```
enum {
    INET_IP     = 0x0,
    INET_TCP    = 0x06,
    INET_UDP    = 0x11
};
```

TCP Options

You can use the options in this section with a protocol level of `INET_TCP`. These TCP options are not association-related. They may be negotiated in all endpoint states except `T_UNBND` and `T_UNINIT`. They are read-only in state `T_UNBND`.

```
#define TCP_NODELAY                             0x01    /* set TCP delay mode */
#define TCP_MAXSEG                              0x02    /* read max segment size */
#define TCP__NOTIFY_THRESHOLD                   0x10    /* reserved */
#define TCP_ABORT_THRESHOLD                     0x11    /* reserved */
#define TCP__CONN_NOTIFY_THRESHOLD              0x12    /* reserved */
#define TCP_CONN_ABORT_THRESHOLD                0x13    /* reserved */
#define TCP_OOBINLINE                           0x14    /* reserved */
#define TCP_URGENT_PTR_TYPE                     0x15    /* reserved */
#define TCP_KEEPALIVE                  OPT_KEEPALIVE    /* activate keep-alive timer */
};
```

Option descriptions

`TCP_NODELAY` Set the TCP delay mode. By default, when TCP has a full segment's worth of data, it sends the segment immediately; but if it receives less than a segment's worth of data and has not yet received acknowledgment for the last packet sent, it saves the data until it either receives a full segment's worth, it receives acknowledgment for the last packet, or until a timeout period has expired. (In this context, a *full segment* is the maximum-sized unit of data that can be sent by TCP at one time and a *packet* is data that is transmitted as a single unit.) Specify `T_YES` for this option to cause all data to be sent immediately. Specify `T_NO` to return TCP to the default delay mode. A request to set this option to no delay is an absolute requirement.

`TCP_MAXSEG` Read the maximum TCP segment size. The maximum segment size is returned as an unsigned long specifying the number of octets. This option is read-only.

`TCP_NOTIFY_THRESHOLD`
Reserved.

`TCP_ABORT_THRESHOLD`
Reserved.

`TCP_CONN_NOTIFY_THRESHOLD`
Reserved.

`TCP_CONN_ABORT_THRESHOLD`
Reserved.

`TCP_OOBINLINE` Reserved.

`TCP_URGENT_PTR_TYPE`
Reserved.

`TCP_KEEPALIVE` Activate the keep-alive timer. If this option is set on, TCP monitors idle connections and sends a keep-alive packet to check a connection after a preset time has expired. You use a `t_kpalive` structure, described later in this section, to specify the value of this option. The default state for the keep-alive timer is off. A request to activate or deactivate the keep-alive timer is an absolute requirement.

The `TCP_KEEPALIVE` option uses a `t_kpalive` structure, defined as follows:

```
struct t_kpalive {
    long    kp_onoff;       /* option on/off */
    long    kp_timeout;     /* timeout in minutes */
};
```

Field descriptions

`kp_onoff` Activate or deactivate the keep-alive timer. Set this field to `T_YES` to activate the timer or to `T_NO` to deactivate it. A request to activate or deactivate the timer is an absolute requirement. The default value of this field is `T_NO`. The Open Transport TCP implementation does not support the value `T_YES|T_GARBAGE` for this field.

`kp_timeout` Set the requested timeout value, in minutes. Specify a value of `T_UNSPEC` to use the default value. You may specify any positive value for this field of 120 minutes or greater.

The timeout value is not an absolute requirement; if you specify a value less than 120 minutes, TCP will renegotiate a timeout of 120 minutes.

UDP Options

You can use the options in this section with a protocol level of INET_UDP. The UDP_CHECKSUM option is association-related. It may be negotiated in all endpoint states except T_UNBND and T_UNINIT. It is read-only in state T_UNBND. The UDP_RX_ICMP option is read-only in all states.

```
#define UDP_CHECKSUM            OPT_CHECKSUM        /* calculate checksum */
#define UDP_RX_ICMP             0x2                 /* read max segment size */
```

Option descriptions

UDP_CHECKSUM — Activate or deactivate a checksum calculation. Specify T_YES to activate the checksum calculation or T_NO to deactivate it. The default value for this option is T_YES. If this option is returned by the OTRcvUData function, its value indicates whether a checksum was present in the received datagram. UDP discards packets that do not have valid checksums when this option is activated. UDP relies on checksum calculations to provide reliable data delivery; under normal circumstances, you should never deactivate this option. A request to activate or deactivate checksums is an absolute requirement.

UDP_RX_ICMP — Determine whether the UDP Streams module has received an ICMP message. This option returns a Boolean value.

IP Options

You can use the options in this section with a protocol level of INET_IP. The IP_OPTIONS and IP_TOS options are association-related; the other IP options are not. The IP_REUSEADDR option may be negotiated in all endpoint states except T_UNINIT. The other options may be negotiated in all endpoint states except T_UNBND and T_UNINIT. They are read-only in state T_UNBND. A request for any of these options is an absolute requirement.

```
#define IP_OPTIONS                  0x01    /* enable/disable options */
#define IP_TOS                      0x02    /* set/get type of service */
#define IP_TTL                      0x03    /* set/get time to live */
#define IP_REUSEADDR                0x04    /* bind multiple addresses to one port */
#define IP_DONTROUTE                0x10    /* bypass standard routing */
#define IP_BROADCAST                0x20    /* get permission to send broadcasts */
#define IP_HDRINCL                  0x1002  /* reserved */
#define IP_RCVOPTS                  0x1005  /* reserved */
#define IP_RCVDSTADDR               0x1007  /* reserved */
#define IP_MULTICAST_IF             0x1010  /* set/get IP multicast interface */
#define IP_MULTICAST_TTL            0x1011  /* set/get multicast time to live */
#define IP_MULTICAST_LOOP           0x1012  /* set/get IP multicast loopback */
#define IP_ADD_MEMBERSHIP           0x1013  /* add an IP group membership */
#define IP_DROP_MEMBERSHIP          0x1014  /* drop an IP group membership */
#define IP_BROADCAST_IF             0x1015  /* reserved */
#define IP_RCVIFADDR                0x1016  /* reserved */
```

Possible flag values for the `IP_TOS` option are as follows:

```
/* IP_TOS precedence levels */
enum {
    T_ROUTINE           = 0,
    T_PRIORITY          = 1,
    T_IMMEDIATE         = 2,
    T_FLASH             = 3,
    T_OVERRIDEFLASH     = 4,
    T_CRITIC_ECP        = 5,
    T_INETCONTROL       = 6,
    T_NETCONTROL        = 7
};

/*  IP_TOS type of service */
enum {
    T_NOTOS             = 0x0,
    T_LDELAY            = (1<<4),
    T_HITHRPT           = (1<<3),
    T_HIREL             = (1<<2)
};
```

Option descriptions

`IP_OPTIONS` Set the value of the Options field in the header of each outgoing IP datagram, or receive the Options field of each incoming IP datagram. This option is intended for use by network debugging and control programs; most applications do not need this option. Normally, you use Open Transport option management functions or configuration strings to set options. The option management functions are described in the chapter "Option Management" in this book, and configuration strings are described in the chapter "Configuration Management" in this book.

The value for this option consists of a string of octets whose formats follow the definitions of IP options in the current RFCs with one exception: If you specify a source routing option, the first address in the list of gateways must be for the first-hop gateway. Open Transport extracts the first-hop gateway address from the option list and adjusts the size of the list before transmitting the packet. The Options field can contain up to 40 octets.

To disable this option, specify an option header only with no option values. This option is enabled by default any time you use an Open Transport option-management function or a configuration string to set an IP option that must be negotiated.

If you enable `IP_OPTIONS`, the function `OTOptionManagement` with the `T_CURRENT` action flag set returns the list of IP options that are currently being sent with outgoing IP datagrams.

The functions `OTConnect` (in synchronous mode only), `OTListen`, `OTRcvConnect`, and `OTRcvUData` return the Options field of the received IP datagram. The `OTRcvUDErr` function returns the Options field of the previously sent datagram that caused the error.

`IP_TOS` Set the Type of Service field of each outgoing IP datagram, or receive the Type of Service field of each incoming IP datagram. Open Transport hosts and routers ignore the Type of Service field, but you can set this value for use with other networks if you so desire. The data for this option is any combination of a Precedence flag and a Type

of Service flag. Use the `OR` operator to combine the flags. The possible values for these flags are shown at the beginning of this section.

If you enable `IP_TOS`, the function `OTOptionManagement` with the `T_CURRENT` action flag set returns the Type of Service flags that are currently being sent with outgoing IP datagrams.

The functions `OTConnect` (in synchronous mode only), `OTListen`, `OTRcvConnect`, and `OTRcvUData` return the Type of Service field of the received IP datagram. The function `OTRcvUDErr` returns the Type of Service field of the previously sent datagram that caused the error.

`IP_TTL` Set the Time to Live field of each outgoing IP datagram. Specify the number of hops as an unsigned char. Each router that processes the datagram decrements the Time to Live field and discards the datagram if the value reaches 0. The default value for this field is 255. Because this is not an association-related option, there is no function that returns the Time to Live field of an incoming datagram.

`IP_REUSEADDR` Allow multiple addresses with the same port number. Set this option to `T_YES` to allow TCP to bind a transport endpoint to a wildcard address (that is, an address of 0) that includes a port number plus bind one or more additional endpoints to distinct fully specified internet addresses that include the same port number. If this option is set to `T_NO` (the default), TCP cannot bind two or more transport endpoints to addresses that include the same port number.

`IP_DONTROUTE` Use addresses on connected subnets only. Set this option to `T_YES` to cause outgoing messages to be delivered to the local network only and not to go through any routers or gateways. (This options sets the time-to-live value to 1.) This option is intended for testing and development purposes. Specify `T_NO` to disable this option. This option is disabled by default.

`IP_BROADCAST` Request permission to send broadcast datagrams. Set this option to `T_YES` to request permission to send broadcast datagrams. Specify `T_NO` to disable this option. This option is disabled by default.

IP_HDRINCL	Include the IP header with received data. Set this option to T_YES to cause RawIP to include the IP header when you read data. Set the option to T_NO (the default) to receive only the data without the header. This option works with the RawIP interface only.
IP_RCVOPTS	Include IP-level options when you call the OTRcvUData function. If you set this option to T_YES (the default), the OTRcvUData function returns IP-level options along with the UDP options when you are receiving UDP data. If you set this option to T_NO, you receive only UDP options.
IP_RCVDSTADDR	For multihomed systems, include with received data the address of the interface on which a message was received. If you specify this option to T_YES, the OTRcvUData function includes the address of the interface. If you specify this option to T_NO (the default), you receive only the data.
IP_MULTICAST_IF	Specify the TCP/IP interface to use for outgoing multicast IP datagrams, or retrieve the interface this option is set to. Specify the interface as a long (for example, if the address is "1.2.3.4", specify the address as 0x01020304). This option and the other multicast options can be used with UDP and RawIP only. In the case that a host is multihomed, this option lets you specify which network interface to use for multicasts. Whereas only one network interface can be used at a time for multicast transmissions, an application can join the same multicast group address on more than one network interface. If you joined the same multicast address on more than one network, this option lets you determine over which network the datagram arrived.
IP_MULTICAST_TTL	Set the Time to Live field for outgoing multicast IP datagrams, or retrieve the Time to Live field set for an interface. Each router that processes the datagram decrements the Time to Live field and discards the datagram if the value reaches 0. Specify the time to live as an unsigned char. To avoid unneccessary network traffic, you should set this value as low as possible. The default value is 1.
IP_MULTICAST_LOOP	Enable loopbacks for outgoing multicast IP datagrams. Set this option to T_YES to cause an outgoing multicast

datagram to be delivered to yourself; set this option to `T_NO` to disable loopbacks. Loopbacks are enabled by default.

`IP_ADD_MEMBERSHIP`

Add a membership in an IP multicast group. You use a `TIPAddMulticast` structure to specify the address and network interface of the group you wish to join. The `TIPAddMulticast` structure is described in "IP Multicast Address Structure" on page 8-28.

`IP_DROP_MEMBERSHIP`

Drop membership in an IP multicast group. You use a `TIPAddMulticast` structure to specify the address and network interface of the group you wish to leave.

`IP_BROADCAST_IF` Reserved.

`IP_RCVIFADDR` Reserved.

The following IP-level options are reserved for use by Apple Computer, Inc.

```
#define     DVMRP_INIT          0x64
#define     DVMRP_DONE          0x65
#define     DVMRP_ADD_VIF       0x66
#define     DVMRP_DEL_VIF       0x67
#define     DVMRP_ADD_LGRP      0x68
#define     DVMRP_DEL_LGRP      0x69
#define     DVMRP_ADD_MRT       0x6A
#define     DVMRP_DEL_MRT       0x6B
```

Functions

This section describes the functions provided by the TCP/IP service provider. In addition to these functions, you need the functions described in the chapter "Endpoints" of this book in order to implement TCP/IP communications.

Opening a TCP/IP Service Provider

This section describes the two functions you can use to open the TCP/IP service provider: `OTAsyncOpenInternetServices` and `OTOpenInternetServices`.

OTAsyncOpenInternetServices

Opens the TCP/IP service provider and returns an internet services reference. This function runs asynchronously.

C INTERFACE

```
OSStatus OTAsyncOpenInternetServices (OTConfiguration *cfig,
                                      OTOpenFlags oflag
                                      OTNotifyProcPtr proc
                                      void *contextPtr);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`cfig`	A pointer to a network configuration structure. Specify `nil` for this parameter to have the function provide the network configuration structure for you. You can also obtain this pointer by using the constant `kDefaultInternetServicesPath` for this parameter.
`oflag`	Reserved. Must be set to 0.
`proc`	A pointer to your notifier function. The TCP/IP service provider passes the internet services reference to your notifier function in the `cookie` parameter.
`contextPtr`	A pointer for your use. The TCP/IP service provider passes this value unchanged to your notifier function.

DESCRIPTION

You must open the TCP/IP service provider before calling any TCP/IP services function other than the address utility functions. You must provide the internet services reference when calling any of these non-utility functions. The `OTAsyncOpenInternetServices` function also sets the mode of all other TCP/IP service provider functions as asynchronous.

If you want to set an option as part of the configuration string, you should translate the option's constant name, given in the header files, into a string that the configuration functions can parse. For the TCP/IP options, Table 8-2 provides the constant name used in "Options," beginning on page 8-29 and the value to use in the configuration string.

Table 8-2 Configuration strings for TCP/IP options

Constant name	Configuration string value
IP_OPTIONS	"Options"
IP_TOS	"TOS"
IP_TTL	"TTL"
IP_RCVDSTADDR	"RcvDestAddr"
IP_RCVIFADDR	"RcvIFAddr"
IP_RCVOPTS	"RcvOPts"
IP_REUSEADDR	"ReuseAddr"
IP_DONTROUTE	"DontRoute"
IP_BROADCAST	"Broadcast"
IP_HDRINCL	"HdrIncl"
IP_MULTICAST_IP	"MulticastIF"
IP_MULTICAST_TTL	"MulticastTTL"
IP_MULTICAST_LOOP	"MulticastLoop"
IP_ADD_MEMBERSHIP	"AddMembership"
IP_DROP_MEMBERSHIP	"DropMembership"
IP_BROADCAST_IF	"BroadcastIF"
UDP_CHECKSUM	"Checksum"
UDP_RX_ICMP	"RxICMP"
TCP_NODELAY	"NoDelay"
TCP_OOBINLINE	"OOBInline"

continued

Table 8-2 Configuration strings for TCP/IP options (continued)

Constant name	Configuration string value
TCB_MAXSEG	"MaxSeg"
TCP_NOTIFY_THRESHOLD	"NotifyThreshold"
TCP_ABORT_THRESHOLD	"AbortThreshold"
TCP_CONN_NOTIFY_THRESHOLD	"ConnNotifyThreshold"
TCP_CONN_ABORT_THRESHOLD	"ConnAbortThreshold"
TCP_KEEPALIVE	"KeepAlive"

COMPLETION EVENT CODES

T_OPENCOMPLETE	0x20000007	The OTAsyncOpenInternetServices function has completed.

SEE ALSO

The OTOpenInternetServices function (described next) is a synchronous version of the TCP/IP open services function.

The network configuration structure and OTCreateConfiguration function are described in the chapter "Configuration Management" in this book.

Use the OTCloseProvider function, described in the chapter "Endpoints" in this book, to close a TCP/IP service provider when you are finished using it.

OTOpenInternetServices

Opens the TCP/IP service provider and returns an internet services reference. This function runs synchronously.

C INTERFACE

```
InetSvcRef OTOpenInternetServices (OTConfiguration *cfig,
                                   OTOpenFlags oflag,
                                   OSStatus *err);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`cfig`	A pointer to a network configuration structure. Specify `nil` for this parameter to have the function provide the network configuration structure for you. You can also obtain this pointer by using the constant `kDefaultInternetServicesPath` for this parameter.
`oflag`	Reserved. Must be set to 0.
`err`	The function result.

DESCRIPTION

You must open the TCP/IP service provider before calling any TCP/IP service function other than the address utility functions. The return value of this function is the internet services reference. You must provide the internet services reference when calling any of these non-utility functions. The `OTOpenInternetServices` function also sets the mode of all other TCP/IP service functions as synchronous.

If you want to set an option as part of the configuration string, you should translate the option's constant name, given in the header files, into a string that the configuration functions can parse. For the TCP/IP options, Table 8-2 on page 8-39 provides the constant name used in "Options," beginning on page 8-29 and the value to used in the configuration string

SEE ALSO

The `OTAsyncOpenInternetServices` function (page 8-38) is an asynchronous version of the TCP/IP open services function.

The network configuration structure and `OTCreateConfiguration` function are described in the chapter "Configuration Management" in this book.

Use the `OTCloseProvider` function, described in the chapter "Endpoints" in this book, to close a TCP/IP service provider when you are finished using it.

Resolving Internet Addresses

This section describes the functions that provide access to the services of the domain name resolver (DNR).

OTInetStringToAddress

Resolves a domain name to its equivalent internet addresses.

C INTERFACE

```
OSStatus OTInetStringToAddress (InetSvcRef ref,
                                char *name,
                                InetHostInfo *hinfo);
```

C++ INTERFACE

```
OSStatus TInternetServices::StringToAddress (char *name,
                                             InetHostInfo *hinfo);
```

PARAMETERS

`ref` The internet services reference you obtained when you opened the TCP/IP service provider.

`name` A pointer to the domain name you want to resolve. This can be a host name, a partially qualified domain name, a fully qualified domain name, or an internet address in dotted-decimal format.

`hinfo`	A pointer to an `InetHostInfo` structure that you provide. When the function completes, it places the canonical name and up to ten associated IP addresses in this structure. If the function finds less than ten IP addresses, it fills in the rest of the address array with zeros.

DESCRIPTION

Because the architecture of Open Transport TCP/IP provides for multihoming, a single host can be associated with multiple internet addresses. You can use the `OTInetStringToAddress` function to return multiple addresses for multihomed hosts.

Note
Because multihoming has not been implemented in the initial release of Open Transport, the `OTInetStringToAddress` function never returns more than one address. ◆

If you specify an internet address in dotted-decimal format for the `hinfo` parameter, the `OTInetStringToAddress` function places that address in the `InetHostInfo.name` field instead of a canonical name.

If you call the `OTInetStringToAddress` function asynchronously, the TCP/IP service provider calls your notifier function with the `T_DNRSTRINGTOADDRCOMPLETE` completion event code when the function completes. The `cookie` parameter to the notifier function contains the pointer you specified in the `hinfo` parameter. If you had more than one simultaneous outstanding call to the `OTInetStringToAddress` function, you can use this information to determine which call has completed execution.

SPECIAL CONSIDERATIONS

If you call the `OTInetStringToAddress` function asynchronously, do not write to the `InetHostInfo` structure until the function completes.

COMPLETION EVENT CODES

T_DNRSTRINGTOADDRCOMPLETE	0x10000001	The OTInetStringToAddress function has completed.

SEE ALSO

The OTLookupName function (page 8-20) provides a mapper interface to the domain name resolver (DNR) that maps a name to a single internet address.

You can use the DNSAddress structure (page 8-24) to provide a domain name directly to the OTConnect, OTSndUData, and OTResolveAddress functions. The OTConnect, OTSndUData, and OTResolveAddress functions are described in the chapter "Endpoints" in this book.

Use the OTInetAddressToName function (described next) to convert an IP address into a domain name.

The InetHostInfo structure is described in "Internet Host Information Structure" on page 8-27.

You can use the OTInetHostToString function (page 8-57) to convert addresses in InetHost format into character strings using dotted-decimal notation.

OTInetAddressToName

Determines the canonical name of the host associated with an internet address.

C INTERFACE

```
OSStatus OTInetAddressToName (InetSvcRef ref,
                              InetHost addr,
                              InetDomainName name);
```

C++ INTERFACE

```
TInternetServices::AddressToName (InetHost addr,
                                  InetDomainName name);
```

PARAMETERS

`ref`	The internet services reference you obtained when you opened the TCP/IP service provider.
`addr`	The IP address for which you want to determine the associated domain name in either dotted-decimal notation (for example, "12.13.14.15") or hexadecimal notation (for example, "0x0c0d0e0f").
`name`	A character array you must allocate into which the function places the canonical name.

DESCRIPTION

If you call this function asynchronously, the TCP/IP service provider calls your notifier function with the `T_DNRADDRTONAMECOMPLETE` completion event code when the function completes. The `cookie` parameter to the notifier function contains a pointer to the `InetHost` structure you specified in the `addr` parameter. If you had more than one simultaneous outstanding call to the `OTInetAddressToName` function, you can use this information to determine which call has completed execution.

COMPLETION EVENT CODES

`T_DNRADDRTONAMECOMPLETE`	0x10000002	The `OTInetAddressToName` function has completed.

SEE ALSO

Use the `OTInetStringToAddress` function (page 8-42) to determine the addresses associated with an IP domain name.

You can use the `OTInetStringToHost` function (page 8-56) to put the address in `InetHost` format.

Getting Information About an Internet Host

This section describes the functions you can use to get information about an internet host.

OTInetSysInfo

Returns details about a host's processor and operating system.

C INTERFACE

```
OSStatus OTInetSysInfo (InetSvcRef ref,
                        char *name,
                        InetSysInfo *sysinfo);
```

C++ INTERFACE

```
OSStatus TInternetServices::SysInfo (char *name,
                                     InetSysInfo *sysinfo);
```

PARAMETERS

`ref`	The internet services reference you obtained when you opened the TCP/IP service provider.
`name`	The name of the host about which you want information. This can be a host name (including the local host), a partially qualified domain name, or a fully qualified domain name.
`sysinfo`	A pointer to an `InetSysInfo` structure. You must allocate this structure. The function fills it in with the processor type and operating-system version of the host.

DESCRIPTION

The information returned by this function is maintained by the domain name server. If you call this function asynchronously, the TCP/IP service provider calls your notifier function with the `T_DNRSYSINFOCOMPLETE` completion event code when the function completes. The `cookie` parameter to the notifier function contains a pointer to the `InetSysInfo` structure you specified in the `sysinfo` parameter. If you had more than one simultaneous outstanding call to the `OTInetSysInfo` function, you can use this information to determine which call has completed execution.

SPECIAL CONSIDERATIONS

If you call this function asynchronously, do not write to the `InetSysInfo` structure until the function completes.

COMPLETION EVENT CODES

`T_DNRSYSINFOCOMPLETE`	0x10000003	The `OTInetSysInfo` function has completed.

SEE ALSO

The `InetSysInfo` structure is described in "Internet System Information Structure" on page 8-28.

OTInetMailExchange

Returns mail-exchange-host names and preference information for a domain name you specify.

C INTERFACE

```
OSStatus OTInetMailExchange (InetSvcRef ref,
                             char *name,
                             UInt16 *num,
                             InetMailExchange *mx);
```

C++ INTERFACE

```
OSStatus TInternetServices::MailExchange (char *name,
                                          UInt16 *num,
                                          InetMailExchange *mx);
```

PARAMETERS

`ref`	The internet services reference you obtained when you opened the TCP/IP service provider.

`name`	A pointer to a host name, partially qualified domain name, or fully qualified domain name for which you want mail exchange information.
`num`	A pointer to the number of elements in the array pointed to by the `mx` parameter. When the function completes, it sets the number pointed to by the `num` parameter to the actual number of elements filled in.
`mx`	A pointer to the first element in an array of `InetMailExchange` structures. You must allocate the structures in this array.

DESCRIPTION

In order to deliver mail, a mail application must determine the fully qualified domain name of the host to which the mail should be sent. That host might be the final destination of the mail, a mail server, or a router. The domain name system servers maintain mail-exchange resource records that pair domain names with the hosts that can accept mail for that domain. Each domain name can be paired with any number of host names; each record containing such a pair also contains a preference number. The mailer sends the mail to the host with the lowest preference number first and tries the others in turn until the mail is delivered or until the mailer decides that the mail is undeliverable.

The `OTInetMailExchange` function returns mail-exchange-host and preference information for the domain name you specify. You must then determine the address of the host and how best to deliver the mail. You can specify as many elements to the array of `InetMailExchange` structures as you wish.

If you call this function asynchronously, the TCP/IP service provider calls your notifier function with the `T_DNRMAILEXCHANGECOMPLETE` completion event code when the function completes. The `cookie` parameter to the notifier function contains the array pointer you specified in the `mx` parameter. If you had more than one simultaneous outstanding call to the `OTInetMailExchange` function, you can use this information to determine which call has completed execution.

SPECIAL CONSIDERATIONS

If you call this function asynchronously, do not write to the `InetMailExchange` array until the function completes.

COMPLETION EVENT CODES

`T_DNRMAILEXCHANGECOMPLETE`	0x10000004	The `OTInetMailExchange` function has completed.

SEE ALSO

The `InetMailExchange` structure is described in "Internet Mail Exchange Structure" on page 8-29.

Internet mail routing and mail-exchange resource records are described in Request for Comments 974: *Mail Routing and the Domain System*.

Retrieving DNS Query Information

This section describes the function that permits generic domain name service (DNS) queries.

OTInetQuery

Returns DNS query information.

C INTERFACE

```
OSStatus OTInetQuery(InetSvcRef ref, char* name, UInt16 qClass,
                     UInt16 qType, char* buf, size_t buflen,
                     void** argv, size_t argvlen, OTFlags flags);
```

C++ INTERFACE

```
OSStatus TInternetServices::Query(char* name, UInt16 qClass,
                     UInt16 qType, char* buf, size_t buflen,
                     void** argv, size_t argvlen, OTFlags flags)
```

PARAMETERS

`ref`	The internet services reference you obtained when you opened the TCP/IP service provider.
`name`	A pointer to the fully qualified domain name or IP address for which you are asking the query.
`qClass`	The DNS query class, such as `Inet` and `Hesiod`, for which you wish to query.
`qType`	The DNS query type, such as `CNAME` and `PTR`, for which you wish to make a query.
`buf`	A pointer to the buffer in which to store one or more DNS query information structures (`DNSQueryInfo`). Open Transport fits as many complete structures into the buffer as it can; incomplete structures are not returned.
`buflen`	The size (in bytes) of the buffer.
`argv`	A pointer to an empty pointer array that Open Transport can use to return a set of pointers to the individual DNS query information structures returned. This parameter is optional, specify a null pointer if you don't want to use this array.
`argvlen`	The requested length of the `argv` buffer. On return, Open Transport updates this with the actual number of entries in the `argv` array. This parameter is optional; if you specify a null pointer for the `argv` parameter, this length is not returned.
`flags`	Reserved. Set to 0.

DESCRIPTION

The `OTInetQuery` function allows you to use the Domain Name Resolver (DNR) for generic domain name service (DNS) queries. You can ask for any query type and class, and Open Transport returns as many responses as it can fit in the buffer you provide.

The `argv` and `argvlen` parameters are optional. If provided, Open Transport uses the `argv` buffer to return pointers to the locations of individual answers written into the answer buffer pointed to by the `buf` parameter. For example, if you set `argvlen` to 5 and your query receives three answers, `argvlen` would be changed to 3, the value of `argv[0]` would be a pointer to the first answer in the answer buffer, the value of `argv[1]` would be a pointer to the second answer,

the value of `argv[2]` would be a pointer to the third answer, and the rest of the `argv` array would have null pointers.

If you call `OTInetQuery` asynchronously, Open Transport calls your notifier with a `T_DNRQUERYCOMPLETE` event when the call completes. Asynchronous mode is preferred. When using asynchronous mode, you must not touch the `buf` or `argv` structures before the function completes.

The `OTInetQuery` function works with both known and unknown query classes and types. Open Transport expands compressed answers for the `Inet` query class and known query types before returning them into the answer buffer. Answers that are resource records of unknown class and type are put into the answer buffer unparsed because Open Transport assumes that DNS compression is not used.

Explicit simplified functions are provided for the most commonly made queries such as name-to-address (`A`), address-to-name (`PTR`), system CPU and OS (`HINFO`), and mail exchange (`MX`) queries. These are the `OTInetStringToAddress`, `OTInetAddressToName`, `OTInetSysInfo`, and `OTInetMailExchange` functions, respectively. For several basic query types, these functions may be easier to use. The information obtained is the same using either type of function, although in some cases the simplified functions limit the maximum number of answers that can be returned.

Currently, only answers of type `PTR`, `A`, and `CNAME` (name-to-address translations) are cached by OpenTransport. Also, OpenTransport does not currently use this cached information to resolve address-to-name translations because doing so would defeat some existing server load balancing schemes in operation today.

COMPLETION EVENT CODES

`T_DNRQUERYCOMPLETE`	0x10000005	The `OTInetQuery` function has completed.

SEE ALSO

Use the `OTInetStringToAddress` function for a simple name-to-address query (page 8-42), `the OTInetAddressToName` function for a simple address-to-name query (page 8-44), `OTInetSysInfo` function for a system CPU and OS query (page 8-46), and `OTInetMailExchange` function for a mail exchange query (page 8-47).

Address Utilities

The functions described in this section fill in address structures and manipulate domain name strings. They do not involve calls to the domain name resolver and cannot be executed asynchronously.

OTInetGetInterfaceInfo

Returns internet address information about the local host.

C INTERFACE

```
OSStatus OTInetGetInterfaceInfo (InetInterfaceInfo *info,
                                 SInt32 val;
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`info` A pointer to an `InetInterfaceInfo` structure. You must allocate this structure. The function fills in such information as the internet address, subnet mask, broadcast address, MTU, and DNS address for the internet interface indicated by the `index` parameter.

`val` An index into the local host's array of configured IP interfaces. Specify 0 for information about the first interface. Specify `kDefaultInetInterface` to get information about the primary interface.

DESCRIPTION

Because the architecture of Open Transport TCP/IP provides for multihoming, in principle a given host can receive packets simultaneously through more than one network interface. For each IP interface configured for the local host, the

OTInetGetInterfaceInfo function provides the internet address and subnet mask, a default gateway (that is, a gateway, if any exists, that can be used to route any packet to all destinations outside the locally connected subnet), and a domain name server, if any is known. The function also returns the version number of the OTInetGetInterfaceInfo function and, if available, the broadcast address for each interface. If the broadcast address is not available, you can determine it from the internet address and subnet mask.

Note
Because multihoming has not been implemented in the initial release of Open Transport, the OTInetGetInterfaceInfo function never returns information for more than one interface. ◆

SPECIAL CONSIDERATIONS

If Open Transport TCP/IP has not yet been loaded into memory, the OTInetGetInterfaceInfo function returns no valid interfaces. Open Transport TCP/IP is not loaded until a TCP/IP application is running unless the user has selected "TCP always loaded" in the TCP/IP control panel.

The OTInetGetInterfaceInfo function cannot block and always runs synchronously.

SEE ALSO

The InetInterfaceInfo structure is described in "Internet Interface Information Structure" on page 8-26.

See "OTBind" on page 8-16 for information on binding an endpoint to all configured IP interfaces.

OTInitInetAddress

Fills in an InetAddress structure with the data you provide.

C INTERFACE

```
void OTInitInetAddress (InetAddress *addr,
                        InetPort port,
                        InetHost host);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`addr`	A pointer to an `InetAddress` structure that you allocate. The function fills in this structure.
`port`	The port number of the address.
`host`	The IP address of the host in network byte order (that is, high-order byte first) in hexadecimal format. This must be a fully qualified address.

DESCRIPTION

This function fills in the `fAddressType` field of the `InetAddress` structure with the value `AF_INET`. You use the `InetAddress` structure when providing a TCP or UDP address to the Open Transport functions `OTConnect`, `OTSndURequest`, and `OTBind`. You are not required to use the `OTInitInetAddress` function when creating an `InetAddress` structure; this function is provided for your convenience only.

SEE ALSO

The `InetAddress` structure is described in "Internet Address Structure" on page 8-23.

You can use the `OTInetStringToHost` function (page 8-56) to put the address in `InetHost` format.

OTInitDNSAddress

Fills in a `DNSAddress` structure with the data you provide.

C INTERFACE

```
size_t OTInitDNSAddress (DNSAddress *addr,
                         char *str);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`addr`	A pointer to a `DNSAddress` structure that you allocate. The function fills in this structure.
`str`	A pointer to a domain name string. This string can be just a host name (otteam), a partially qualified domain name (for example, “otteam.ssw”), a fully qualified domain name (for example, “otteam.ssw.apple.com.”), or an internet address in dotted-decimal format (for example, “17.202.99.99”), and can optionally include the port number (for example, “otteam.ssw.apple.com:25” or “17.202.99.99:25”).

DESCRIPTION

This function fills in the `fAddressType` field of the `DNSAddress` structure with the value `AF_DNS`, fills in the `fName` field with the address string you specify, and returns the size of the resulting `DNSAddress` structure as an unsigned integer. You can use the `DNSAddress` structure to provide an address when you use a UDP or TCP endpoint. If you do so, the domain name resolver resolves the address for you automatically.

SEE ALSO

The `DNSAddress` structure is described in “DNS Address Structure” on page 8-24.

OTInetStringToHost

Converts an IP address string from dotted-decimal notation or hexadecimal notation to an `InetHost` data type.

C INTERFACE

```
OSStatus OTInetStringToHost (char *str,
                             InetHost *host);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`str`	A pointer to a character string containing an IP address in either dotted-decimal notation (for example, "12.13.14.15") or hexadecimal notation (for example, "0x0c0d0e0f").
`host`	A pointer to the address as an `InetHost` data type. The function allocates storage for this address and returns the pointer to you.

SPECIAL CONSIDERATIONS

The `OTInetStringToHost` function cannot block and always runs synchronously. It does not use the services of the DNR.

SEE ALSO

The `InetHost` data type is defined in "Basic Types and Constants" on page 8-21.

To convert an `InetHost` address into dotted-decimal format, use the `OTInetHostToString` function (described next).

OTInetHostToString

Converts an an address in `InetHost` format into a character string in dotted-decimal notation.

C INTERFACE

```
void OTInetHostToString (InetHost *host,
                         char *str);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`host`	A pointer to the address as an `InetHost` data type.
`str`	A pointer to a C string containing an IP address in dotted-decimal notation (for example, "12.13.14.15"). You must allocate storage for this string and provide the pointer to the function.

SPECIAL CONSIDERATIONS

The `OTInetHostToString` function cannot block and always runs synchronously. It does not use the services of the DNR.

SEE ALSO

The `InetHost` data type is defined in "Basic Types and Constants" on page 8-21.

To convert a string from dotted-decimal notation or hexadecimal notation to an `InetHost` data type, use the `OTInetStringToHost` function (page 8-56).

Introduction to AppleTalk

Contents

This chapter provides an overview of the Open Transport implementation of AppleTalk, Apple Computer's proprietary networking technology. AppleTalk is a communications network system that interconnects Macintosh computer workstations, printers, shared modems, and other computers acting as file servers and print servers. AppleTalk allows these devices to exchange information through communications hardware and software.

Open Transport provides networking functions that you can use to send and receive data across a network, and you can choose to take advantage of Open Transport's transport-independent architecture by using these functions without any protocol-specific options. If, however, you want to take advantage of a particular protocol or protocol family, such as AppleTalk, you can choose specific options that are dependent on a protocol and its implementation.

If you want to use AppleTalk, the specific set of Open Transport functions you call depends on the nature of the specific protocol you use—whether it is connectionless or connection-oriented, and transactionless or transaction-based. For example, you use different functions to send and receive data with a connection-oriented protocol like AppleTalk Data Stream Protocol (ADSP) or Printer Access Protocol (PAP) than with a connectionless protocol like Datagram Delivery Protocol (DDP) or AppleTalk Transaction Protocol (ATP).

Read this chapter if you want an overview of AppleTalk networks and AppleTalk protocols. You can also read this chapter for help in deciding which AppleTalk protocols to use for your application's requirements.

This chapter introduces

- AppleTalk networking in general
- AppleTalk protocols implemented in Open Transport
- AppleTalk service providers
- AppleTalk mappers

This chapter and the other AppleTalk chapters in this book describe how to use AppleTalk-specific options with the Open Transport networking functions that are appropriate for the AppleTalk protocol you wish to use.

Because an AppleTalk network includes both hardware and software, the information in this book constitutes only a small part of the body of literature documenting AppleTalk. An important resource for any AppleTalk network developer is the book *Inside AppleTalk*, second edition, which has detailed specifications for each of the AppleTalk protocols.

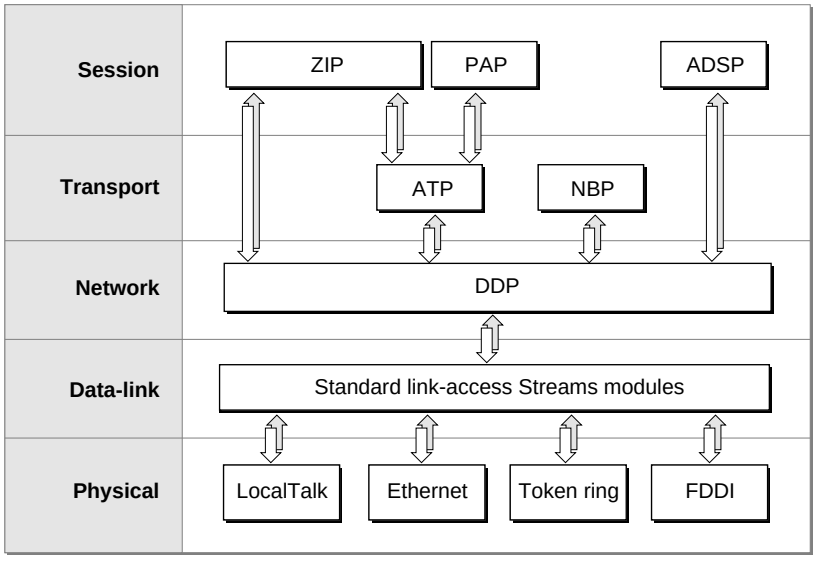
About AppleTalk

Every Macintosh computer includes AppleTalk hardware and software, so if your application needs to communicate with other Macintosh computers, you may want to use an AppleTalk protocol. AppleTalk includes protocols that handle Macintosh workstation-server interaction, LaserWriter and ImageWriter printing, data exchange through data streams or packets, and AppleTalk name lookups across a network.

Although AppleTalk includes protocols that provide connection-oriented services, it is considered a connectionless network because all AppleTalk data is ultimately delivered by the Datagram Delivery Protocol (DDP), which implements connectionless packet delivery. Connection-oriented AppleTalk protocols that establish sessions and provide reliable delivery of data, such as the AppleTalk Data Stream Protocol (ADSP) and the AppleTalk Transaction Protocol (ATP), are built on top of the connectionless packet services that DDP provides. In the AppleTalk protocol stack, each protocol in a specific layer provides a set of functions and services to one or more protocols in a higher-level layer.

The AppleTalk architecture is closely aligned with the industry-standard Open Systems Interconnection (OSI) networking model. Figure 9-1 shows the AppleTalk protocols supported by Open Transport and shows how they relate to one another in the layers defined by the OSI model.

Figure 9-1 AppleTalk protocol stack and the OSI model



Here are some points worth noting about how AppleTalk under Open Transport maps to the OSI model:

- At the session layer, the AppleTalk Data Stream Protocol (ADSP) provides its own stream-based transport layer services that allow for full-duplex dialogs, while the Printer Access Protocol (PAP) uses the transaction-based services of the AppleTalk Transaction Protocol (ATP) to transport workstation commands to servers. The Zone Information Protocol (ZIP) is also at the session layer; a subset of its functions are available through AppleTalk service providers.
- At the transport layer, there are the AppleTalk Transaction Protocol (ATP) and Name-Binding Protocol (NBP), but NBP is accessible only through mapper providers. In addition to these two protocols, ADSP includes functions that span both the session and the transport layers.
- At the network layer, the Datagram Delivery Protocol (DDP) is AppleTalk's network delivery protocol.

- At the data-link layer, various link-access protocols support the underlying networking hardware. Open Transport provides standard Streams modules for the LocalTalk, Ethernet, token ring, and FDDI drivers.

AppleTalk Networks and Addresses

An AppleTalk network can be either a nonextended network or an extended network. Applications can use AppleTalk protocols across a single AppleTalk network or an **AppleTalk internet,** which is a number of interconnected AppleTalk networks. An AppleTalk internet can include a mix of LocalTalk, TokenTalk, EtherTalk, and FDDITalk networks, or it can consist of multiple networks of a single type, such as several LocalTalk networks. An AppleTalk internet can include both nonextended and extended networks.

An AppleTalk **nonextended network** is one in which

- the network has one network number assigned to it
- the network supports only one zone
- all nodes on the network share the same network number and zone name
- each node on the network has a unique node ID

LocalTalk is an example of a nonextended network. Each node on a nonextended network, such as LocalTalk, has a unique 8-bit node ID. Since there are 256 possible combinations of 8 bits, and three IDs are not available (ID 255 is reserved for broadcast messages and ID 0 and 254 are not allowed), a nonextended network can support up to 253 active nodes at a time.

An AppleTalk **extended network** is one in which

- the network has a range of network numbers assigned to it
- the network supports multiple zones
- each node on the network has a unique network number-node ID combination to identify it

Table 9-1 summarizes the identifiers that you use for AppleTalk addressing.

Table 9-1 AppleTalk addressing identifiers

Identifier	Description
Network number	A 16-bit number that identifies the network to which a node is connected. An extended network is defined by a range of network numbers.
Node ID	An 8-bit number that identifies a node.
Zone name	A name assigned to a logical grouping of nodes in an AppleTalk network or internet.
Socket number	An 8-bit number that identifies a socket.
DDP type	An 8-bit number that identifies an endpoint's protocol.

Each network is assigned a **network number** so that an AppleTalk internet router can determine the packet's destination network number and forward the packet through an internet from one router to another until the packet arrives at its correct destination network. An extended network uses a range of network numbers. Nodes on an extended network can have different zone names and different network numbers within the network number range.

A **node** is a data-link addressable entity on an AppleTalk network; all physical devices on an AppleTalk network are nodes. When a node first connects to an AppleTalk network or is rebooted, AppleTalk dynamically assigns it a unique 8-bit **node ID.** For a node on an extended network, AppleTalk also assigns it a 16-bit network number within the range of numbers assigned to the extended network that the device is connected to. Once a packet arrives at its destination network, the packet is delivered to its destination node within that network, based on the node ID.

Note that because AppleTalk assigns node IDs dynamically whenever a node joins the network or is rebooted, a node's address on an AppleTalk network can change from time to time, although a computer attempts to reuse the node ID it last used. NBP provides a mapping of logical names (like those in the Chooser) to physical addresses in such a way that if the node ID changes, you can still find your application. This mapping is discussed further in the chapters "AppleTalk Addressing" and "AppleTalk Service Providers" in this book.

A **zone** is a logical grouping of nodes within an AppleTalk internet. The use of zones allows a network administrator to set up departmental or other logical sets of nodes in an internet. A single extended network can contain nodes

belonging to any number of zones; an individual node on an extended network can belong to only one zone. Each zone is identified by a unique zone name.

A **socket** is an addressable data-link entity on a network. Endpoints exchange data with each other across an internet through sockets. Because each endpoint has its own socket address, a node can have multiple concurrent open connections, for example, one to a file server and one to a printer. A node can have several sockets open at the same time, so each endpoint on an AppleTalk network is associated with a unique 8-bit **socket number.**

AppleTalk sockets are divided into two groups: statically assigned sockets and dynamically assigned sockets. **Statically assigned sockets** are those sockets that are permanently reserved for a designated protocol or process. For example, socket 4 is always reserved as the echo socket, used for echoing packets across a network. **Dynamically assigned sockets** are those sockets arbitrarily assigned by DDP if you do not specify a socket number when binding an endpoint; DDP returns the socket number to you in the endpoint's address when the binding has completed.

In certain situations, you can bind multiple endpoints to a single socket. For connectionless endpoints, each must uses a different protocol. For connection-oriented endpoints, they can all use the same protocol, but each must establish a connection with a different remote endpoint.

Multinodes

AppleTalk's **multinode architecture** allows an application to acquire virtual node IDs, called **multinode IDs.** These multinode IDs allow the computer running your application to appear as multiple nodes on the network even though it is only one physical entity. Each acquired multinode is in addition to the standard node ID already assigned to the computer when it joined the network as a node.The prime example of a multinode application is Apple Remote Access (ARA).

You can use a multinode to receive broadcast packets and any AppleTalk packets addressed to it through its multinode ID. You must then process the packets in a custom manner. A multinode ID is not connected to the AppleTalk protocol stack above the data-link layer, which means that an application that uses a multinode cannot use the services of higher-level protocols such as NBP, ATP, and ADSP, but instead must implement its own higher-level protocols if it expects packets for such protocols.

Handling Miscellaneous Events

In classic AppleTalk, you could use the **AppleTalk Transition Queue (ATQ)** to inform your application of **miscellaneous events** that occurred unexpectedly within AppleTalk. In Open Transport AppleTalk, this facility has been modified to allow your endpoint to receive only a few predefined events. Any applications that rely on the AppleTalk Transition Queue must use AppleTalk backward compatibility to handle them in the classic AppleTalk manner.

In Open Transport AppleTalk, there are five miscellaneous events that you can receive on your endpoint, which does not need to be bound. They are as follows:

Miscellaneous event	Value	Explanation
`T_ATALKROUTERDOWNEVENT`	0x23010051	The router on your application's network is no longer available.
`T_ATALKROUTERUPEVENT`	0x23010052	A router has become available on your application's network.
`T_ATALKZONENAMECHANGEDEVENT`	0x23010053	The router has changed the name for your application's zone.
`T_ATALKCONNECTIVITYCHANGEDEVENT`	0x23010054	An multinode connection was established or disconnected on your network.
`T_ATALKCABLERANGECHANGEDEVENT`	0x23010055	A router has become available on your network, and your endpoint's address is no longer in the correct local-network number range.

To receive these events, your application must use the `OTIoctl` function with a provider reference value, the constant `kOTGetMiscellaneousEvents` as its command, and the value of 1 as its data. For more information on the `OTIoctl` function, refer to the chapter "Providers" in this book.

Configuring AppleTalk Protocol Providers

When you want to use a particular AppleTalk protocol, you open an endpoint configured for that protocol. To do this, you use specific constants as part of a configuration string that you pass to the Open Transport function for opening endpoints. This string specifies to Open Transport how to create the correct endpoint for you. For more information on the functions that you use to open

endpoints, mappers, and AppleTalk service providers, refer to the chapters in this book on the specific type of provider; for more information about configuring providers, see the chapter "Configuration Management" in this book.

Table 9-2 lists which constants to use to configure the AppleTalk providers. Note that these values are subject to change—they are included here only to provide an overview of how OpenTransport configures providers. Be sure to consult the AppleTalk header file for the current values.

Table 9-2 Protocol identifiers for use in configuring AppleTalk providers

Constant	Configuration string value	Type of provider configured
`kNBPName`	"nbp"	NBP mapper provider
`kDDPName`	"ddp"	DDP endpoint provider
`kATPName`	"atp"	ATP endpoint provider
`kADSPName`	"adsp"	ADSP endpoint provider
`kPAPName`	"pap"	PAP endpoint provider

There is one exception to the typical method of configuring providers. AppleTalk service providers do not have a string equivalent value. You configure an AppleTalk service provider with the constant `kDefaultAppleTalkServicesPath`, which has a value of `((OTConfiguration*)-3)`. The code for creating an AppleTalk service provider is as follows:

```
OTOpenEndpoint(kDefaultAppleTalkServicesPath, 0, &err)
```

If you want to set an option as part of the configuration string, you need to know which protocols use which options and how to translate the option's constant name, given in the header files, into a string that the configuration functions can parse. For the AppleTalk options, Table 9-3 provides the constant name, the value used in the configuration string, and the protocols that use that option.

Table 9-3 Indicating AppleTalk options in the configuration string

Constant name	Configuration string value	Valid protocols
OPT_CHECKSUM	"Checksum"	DDP, ATP, ADSP, PAP
OPT_SELFSEND	"SelfSend"	DDP
OPT_ENABLEEOM	"EnableEOM"	ADSP, PAP
OPT_INTERVAL	"RetryInterval"	ATP
OPT_RETRYCNT	"RetryCount"	ATP
ATP_OPT_RELTIMER	"ReleaseTimer"	ATP
PAP_OPT_OPENRETRY	"OpenRetry"	PAP

To configure a provider with an option string, you put the string and its assigned value in parentheses after the protocol that uses it, as in the following lines of code:

```
OTOpenEndpoint(OTCreateConfiguration
                        ("adsp,ddp(Checksum=1),ltlkB"), 0, NULL, &err)

OTOpenEndpoint(OTCreateConfiguration
                        (kADSPName"(EnableEOM=1)"), 0, NULL, &err);
```

About AppleTalk Protocols Under Open Transport

Each of the AppleTalk protocols implements a different set of functions and services, and your choice of which protocol to use depends primarily on your application's needs. For example, if you need a connection-oriented transactionless protocol to exchange data with another endpoint, ADSP is your most likely choice. Open Transport supports most AppleTalk protocols and provides protocol-specific options for various Open Transport functions. Which functions to use with which AppleTalk protocol, and which options are permitted for each, are discussed in this book in the specific chapter for each AppleTalk protocol.

You use most AppleTalk protocols by specifying them explicitly when opening an endpoint. ADSP, ATP, and PAP fall into this category. Because DDP is the network delivery protocol for AppleTalk, you can specify it explicitly or, more often, you use it implicitly when you choose other higher-level AppleTalk protocols.

You don't use NBP and ZIP explicitly with endpoints: NBP-configured mapper providers access NBP to register and delete an application's name as a network-visible entity and to look up other endpoint names on the network; AppleTalk service providers use a subset of ZIP functions to provide applications with information about zones and the current AppleTalk environment.

Note
In order to exchange data and share resources, nodes must be running the same protocol, but they do not all have to be running Open Transport. For example, if one endpoint is using ADSP to send data to an endpoint on another computer, the other endpoint must also be running ADSP, although it does not have to be the Open Transport ADSP implementation. ◆

Open Transport implements two connection-oriented transactionless AppleTalk protocols that you can use to send and receive data: ADSP, and PAP. As discussed in the chapter "Introduction to Open Transport," the decision of which protocol to use is typically based on whether it maintains a connection and uses discrete transactions or sends a stream of data.

Open Transport also implements two connectionless AppleTalk protocols that you can use to send and receive data: ATP and DDP. ATP is a transaction-based protocol and sends request transactions and receives replies; DDP does not send transactions, instead it sends individual packets of data, called *datagrams*, and expects no reply.

The AppleTalk protocols that Open Transport supports for endpoints are shown in Table 9-4.

Table 9-4 Open Transport support for AppleTalk endpoint protocols

	Connectionless	Connection-oriented
Transactionless	DDP	ADSP PAP
Transaction-based	ATP	ASP (See note)

Note
The **AppleTalk Session Protocol (ASP)** is a connection-oriented transaction-based protocol that sets up and maintains sessions between workstations and servers. Apple Computer, Inc. recommends using ADSP instead of ASP for all new application protocol products. Although not currently supported by Open Transport, the next release of Open Transport will include a full implementation of ASP. ◆

In general, applications use ADSP for symmetrical data exchange between two peer endpoints and PAP for printing data. PAP is a client of ATP, so it takes advantage of ATP's reliable data delivery services. Because DDP underlies all AppleTalk data delivery, all AppleTalk protocols ultimately use DDP for data transport.

AppleTalk Addressing and the Name Binding Protocol (NBP)

Because AppleTalk assigns node IDs dynamically whenever a node joins the network or is rebooted, a node's address on an AppleTalk network can change from time to time. Applications cannot assume that the physical address of an AppleTalk endpoint is stable, and therefore a reliable mapping of user names to physical addresses is very important for AppleTalk.

The **Name-Binding Protocol (NBP)** is an AppleTalk protocol that maintains this mapping, and you can access this information through a mapper provider configured for NBP. Because AppleTalk supports dynamic name registration, NBP mapper providers can use the Open Transport name registration and deletion functions as well as the other mapper functions.

In order for you to make the name of your AppleTalk endpoint visible to other applications on a network, you must register its name. There are two ways of doing this, but in either case, Open Transport uses NBP to associate the

endpoint's name with its physical address. Once your application is registered, it is a network-visible entity that other applications can locate.

Through mapper library functions, AppleTalk applications can

- register and delete endpoints as network-visible entities
- look up other endpoint names, using wildcards as needed to match partial names
- initialize name and address structures
- get and set endpoint name information

The chapter "Mappers" in this book describes how to use Open Transport mapper providers and the chapter "AppleTalk Addressing" in this book discusses how to use NBP mapper providers to identify and locate endpoints on a network.

The AppleTalk Service Provider

An **AppleTalk service provider** is an Open Transport provider that gives applications access to information and services that are specific to the AppleTalk protocol stack. Applications use an AppleTalk service provider to obtain zone names and to get information about the current AppleTalk environment for a given machine.

The AppleTalk service provider is able to provide information about zones by implementing a subset of the Zone Information Protocol (ZIP). AppleTalk service provider functions allow applications to query routers for information about

- their own node's zone name
- the names of all the zones on their local network
- the names of all the zones throughout the AppleTalk internet

ZIP is implemented primarily in AppleTalk internet routers, each of which maintains a zone information table that maps the relationships between zone names and network numbers for AppleTalk networks.

The chapter "AppleTalk Service Providers" in this book discusses how to use AppleTalk service providers.

Datagram Delivery Protocol (DDP)

The **Datagram Delivery Protocol (DDP)** is a connectionless transactionless protocol that transfers data between sockets as discrete packets, or *datagrams*, with each packet carrying its destination socket address. DDP attempts to deliver any packet with a valid address but does not inform the sender when it cannot deliver a packet, and it cannot request the sender to retransmit lost or damaged packets. This level of service is referred to as **best-effort delivery.** DDP does not include support to ensure that all sent packets are received at the destination or that those packets that are received are in the correct order. Higher-level protocols that use the services of DDP provide for reliable delivery of data. DDP uses whichever link-access protocol the user selects; that is, DDP can send its datagrams through any type of data link and transport media, provided the network hardware is compatible with Open Transport.

For applications such as games that do not require reliable delivery of data or diagnostic tools that retransmit at regular intervals to estimate averages, DDP suffices. DDP involves less overhead and provides faster performance than higher-level protocols.

The chapter "Datagram Delivery Protocol (DDP)" in this book describes how to use DDP under Open Transport.

AppleTalk Data Stream Protocol (ADSP)

The **AppleTalk Data Stream Protocol (ADSP)** is a connection-oriented transactionless protocol that supports sessions over which applications can exchange full-duplex streams of data. In addition to ensuring reliable delivery of data, ADSP provides a peer-to-peer connection; that is, both ends of the connection can exert equal control over the exchange of data. ADSP also provides an application with a means of sending expedited attention messages to pass control information between the two communicating applications without disrupting the main flow of data.

ADSP appears to its clients to maintain an open pipeline between the two entities at either end. Either entity can write a stream of bytes to the pipeline or read data bytes from the pipeline. However, because ADSP, like all other higher-level AppleTalk protocols, is a client of DDP, the data is actually sent as packets. This allows ADSP to correct transmission errors in a way that would not be possible for a true data stream connection. Thus, ADSP retains many of the advantages of a transaction-based protocol while providing to its clients a connection-oriented full-duplex data stream.

ADSP also includes features that let you authenticate the identity of the party at the other end of the connection and send encrypted data, which is then decrypted at the other end.

Note
The authentication and encryption features of ADSP are referred to as the *AppleTalk Secure Data Stream Protocol (ASDSP)* and are not currently supported in Open Transport, but they will be supported in the next release of Open Transport. ◆

The chapter "AppleTalk Data Stream Protocol (ADSP)" in this book describes how to use ADSP under Open Transport.

AppleTalk Transaction Protocol (ATP)

The **AppleTalk Transaction Protocol (ATP)** is a connectionless transaction-based protocol that allows two endpoints to execute request-and-response transactions. Either ATP endpoint can request another ATP endpoint to perform an action; the other ATP endpoint then carries out the action and transmits a response reporting the outcome. ATP provides reliable delivery of data by ensuring that data packets are delivered in the correct sequence and by retransmitting any packets that are lost.

ATP is useful if your application sends small amounts of data and can tolerate a minor degree of performance degradation. Games that are based on request-and-response dialogs can make efficient use of ATP.

The chapter "AppleTalk Transaction Protocol (ATP)" in this book describes how to use ATP under Open Transport.

Printer Access Protocol (PAP)

The **Printer Access Protocol (PAP)** is an asymmetrical connection-oriented transactionless protocol that enables communication between client and server endpoints, allowing multiple connections at both ends. PAP uses ATP packets to transport the data once a connection is open to the server.

PAP is the protocol that ImageWriter and LaserWriter printers in the AppleTalk environment use for direct printing—that is, when a workstation sends a print job directly to a printer connected to the network instead of using a print

spooler. Open Transport PAP provides a single protocol implementation for all AppleTalk printers that is integrated into the AppleTalk protocol stack.

The chapter "Printer Access Protocol (PAP)" in this book describes how to use PAP under Open Transport.

AppleTalk Addressing

Contents

This chapter describes how to use the AppleTalk address formats to locate an AppleTalk endpoint or to make your endpoint visible to other endpoints across an Open Transport AppleTalk network. Whenever you want to communicate across the network, you need to be able to identify your own local endpoint and the remote endpoint with which you want to communicate. You can use a user name, a physical socket address, or a combination of the two to identify the endpoints. Open Transport provides a specific address format for each of these cases and several utility functions to initialize them.

In order for you to make the name of your AppleTalk endpoint visible to other applications on a network, you must register its name. There are two ways of doing this, but in either case, Open Transport uses the Name Binding Protocol (NBP) to associate the endpoint's name with its physical address. Open Transport provides several utility functions and a specialized data structure, the NBP entity, for more convenient manipulation of NBP names.

This chapter provides information about endpoint and mapper functions that you can use to register a name, to look up name and address information, and to browse for all protocol addresses associated with a name or name pattern.

You should read this chapter if your application uses an AppleTalk networking protocol and you need to

- specify a local or remote address
- register and delete endpoints as network-visible entities
- look up other endpoint names, using wildcards as needed to match partial names
- initialize address structures
- get and set other endpoint name information

Some of these tasks are available through endpoint and mapper functions, which are described in the chapters "Endpoints" and "Mappers" in this book. You should be familiar with the material in those chapters before you read this chapter.

About AppleTalk Addressing

Because AppleTalk assigns node IDs dynamically whenever a node joins the network or is rebooted (although AppleTalk remembers which node ID the node last used and attempts to reclaim it), a node's address on an AppleTalk network can change from time to time. Applications cannot assume that the physical address of an AppleTalk endpoint is stable, and therefore a reliable mapping of user names to physical addresses is very important for AppleTalk.

The **Name-Binding Protocol (NBP)** provides a mapping of logical names (like those in the Chooser) to physical socket addresses in such a way that if the node ID changes, you can continue to reliably identify your application. An endpoint's logical name is its **NBP name,** also sometimes called its *entity name.* You can access information about your endpoint's logical addressing through an NBP-configured mapper provider, which you can also use to locate other endpoints on the network. Because AppleTalk supports dynamic name registration, NBP mapper providers can use the Open Transport name registration and deletion functions as well as the other mapper functions.

When you bind an AppleTalk endpoint, Open Transport associates the endpoint with a protocol address, which can be in one of these formats:

- The **DDP address** supplies the physical network address of an endpoint.
- The **NBP address** supplies the user-friendly NBP name.
- The **combined DDP-NBP address** combines the endpoint's physical network address and its NBP name.
- The **multinode address** supplies the physical network address of a multinode endpoint.

The following several sections discuss each address format in more detail.

Note
Open Transport also provides a generic name format (indicated by the constant `kOTGenericName` and the `OTAddress` structure). However, AppleTalk functions do not validate addresses in this form, so there is no error checking if it is not a valid address. ◆

Using AppleTalk Addressing

This section explains how you use AppleTalk addressing formats to identify an endpoint and how you use various Open Transport AppleTalk functions to

- initialize an address
- compare two DDP addresses
- register the name of your endpoint
- look up names and addresses to find a specific applicaiton or user name
- manipulate NBP names by using NBP entity structures
- initialize NBP entities
- set and extract the name, type, or zone parts of an NBP name

Specifying a DDP Address

The primary address format is the DDP address format, which is the most commonly used. It identifies the physical socket address for your endpoint. Data transmission is fastest for those functions that use this address format because no lookup or conversion is necessary for Open Transport to find the specified physical location. Functions that use the NBP address format, for example, have to look up the mapping of the NBP name to its physical address, and this extra step slows down communications.

Functions such as `OTBind`, `OTGetProtAddress`, and `OTResolveAddress` return an address in this format. DDP addresses use the DDP address structure (defined by the `DDPAddress` data type), which includes the following fields:

Field	Meaning
Address type	The type of address format
Network number	The endpoint's network
Node ID	The endpoint's node
Socket number	The endpoint's socket
DDP type	A DDP endpoint's type of protocol

Permissable values for these fields are given in the section "The DDP Address Structure" on page 10-16. The DDP type field is discussed further in the chapter "Datagram Delivery Protocol" in this book.

When you bind an AppleTalk endpoint, you typically specify a network number of 0 and a node ID of 0, or leave both fields blank. In most cases, the combination of the network number, the node ID, and the socket number creates a unique identifier for any socket in the AppleTalk internet so that AppleTalk's delivery protocol, DDP, can deliver packets to the correct destination.

Since the DDP type field is ignored by all protocols other than DDP, set this field to 0 unless you plan to use the DDP protocol. For more information on DDP types, see the chapter "Datagram Delivery Protocol (DDP)" in this book.

In using Open Transport functions to send or receive data, you use a `TNetbuf` structure to point to a buffer that holds data for a specific Open Transport function. Listing 10-1 shows how you set up the fields of a DDP address and how you set up a `TNetbuf` structure for it.

Listing 10-1 Setting up a DDP Address

```
void DoCreateDDPAddress(TNetbuf *theNetBuf, long net, short node,
                        short socket)
{
    DDPAddress *ddpAddress;

    /* Allocate memory for the DDPAddress structure. */
    ddpAddress = (DDPAddress*) NewPtr(sizeof(DDPAddress));

    /* Set up a DDPAddress structure. */
    ddpAddress->fAddressType            = AF_ATALK_DDP;
    ddpAddress->fNetwork                = net;
    ddpAddress->fNode                   = node;
    ddpAddress->fSocket                 = socket;
    ddpAddress->fDDPType                = 0;
    ddpAddress->fPad                    = 0;

    /* Set the TNetbuf to point to it. */
    theNetbuf->len              = sizeof(DDPAddress);
    theNetbuf->maxlen           = sizeof(DDPAddress);
    theNetbuf->buf              = (void*)ddpAddress;
}
```

Specifying an NBP Address

You can use the NBP address format to identify an endpoint when you know the user-defined name of an endpoint but not its physical address. Applications that run on an Open Transport AppleTalk network can display these user-friendly NBP names to users while using the DDP addresses internally to locate and address entities. See the section "Looking Up Names and Addresses," beginning on page 10-11 for more information on how Open Transport translates an NBP name into a physical address.

The NBP address format is defined by the NBP address structure, which includes the following fields:

Field	Meaning
Address type	The type of address format
NBP name buffer	A pointer to a text string giving the endpoint's NBP name

The values for these fields are discussed more fully in the section "The NBP Address Structure" on page 10-17. Note that NBP addresses do not "know" the length of the NBP name you might use. To find out its length, you must use the `OTResolveAddress` or `OTLookupName` functions.

An **NBP name** consists of these three fields: **name, type,** and **zone.** The value for each of these fields is an alphanumeric string of up to 32 characters. The NBP name is not case sensitive. When you bind an endpoint with an NBP address, you must specify a value for the name and type fields, but you don't have to specify the zone. The NBP name string is not null-terminated and is in the form

name:type@zone

The name field typically identifies the user of the system or, in the case of a server, the system itself. For example, you could use the name field to register a serial number for your application and then use a mapper provider to search to see if this matches any other NBP name. If it does, your application could branch to some specialized code to prevent the duplicate application from launching or to issue a warning message as part of a software protection scheme.

The type field generally identifies the type of service that the entity provides, for example, "Mailbox" for an electronic mailbox on a server. Applications offering similar services can find one another and identify potential partners by

looking up only those addresses with a specific type. You could request the mapper provider to return the names of all of the registered entities of a certain type, for example, all compiler servers and laser printers.

The zone field identifies the zone within the network to which the node belongs. To indicate the current zone (or no zone, as in the case of a simple network configuration not divided into zones), you can leave this field blank (the preferred method) or you can specify an asterisk (*). To Open Transport, these two methods are equivalent; thus, the strings "MyName:MBOX@*" and "MyName:MBOX" identify the same zone. There are several functions for getting zone information; these are described in the chapter "AppleTalk Service Providers" in this book.

When you use an NBP structure to define an NBP address format, you copy the string specifying the NBP name into the NBP name buffer.

You can use the backslash (\) character in an NBP name to include the colon (:), at sign (@), and the backslash (\) characters in the name. For example, if you wanted to use the name "My\Machine," the type "My:Server" and the zone "My@Zone," you would express it in the following way:

```
My\\Machine:My\:Server@My\@Zone
```

The maximum size of the NBP name buffer is currently defined to be 105 bytes. This permits a string whose name, type, and zone fields each contain the maximum 32 characters, plus 2 bytes for the separator characters (: and @) and 7 bytes for escape characters—that is, combinations of backslash-colon (\:), backslash-at sign (\@), or backslash-backslash (\\).

If you specify an NBP address structure when binding an endpoint, Open Transport assigns a dynamic socket number to the DDP address of the endpoint (because the NBP address cannot supply any socket number) and registers the NBP name you specified for your application.

Listing 10-2 shows how you set up the fields of an NBP address. The statements used to set the size of the `len` and `maxlen` fields of the `TNetbuf` structure simply add the size of the two fields of the NBP address structure: the size of the constant name plus the length of the string equals the length of data stored in the buffer.

Listing 10-2 Setting up an NBP address

```
void DoCreateNBPAddress(TNetbuf *theNetBuf, char* nbpName)
{
    NBPAddress *nbpAddress;
    short nbpSize;

    /* Allocate memory for an NBP structure. */
    nbpSize = sizeof(OTAddressType) + strlen(nbpName);
    nbpAddress = (NBPAddress*) NewPtr(nbpSize);

    /* Set up an NBPAddress structure. */
    nbpAddress->fAddressType            = AF_ATALK_NBP;
    strcpy(nbpAddress->fNBPNameBuffer, nbpName);

    /* Set the TNetbuf to point to it. */
    theNetbuf->len              = nbpSize;
    theNetbuf->maxlen           = nbpSize;
    theNetbuf->buf              = (void*)nbpAddress;
}
```

Specifying a Combined DDP-NBP Address

You use the the combined DDP-NBP address format when you want to bind an endpoint with a specific NBP name to a specific socket. As the name suggests, this format combines the DDP address and the NBP address. Its data structure begins, as do all of the address structures, with a constant defining which address format to use; then it includes all the standard DDP address fields and ends with the standard NBP name buffer field. See the previous two subsections, “Specifying a DDP Address” and “Specifying an NBP Address,” and the section “The Combined DDP-NBP Address Structure” on page 10-18 for discussion of these fields, and also refer to *Inside AppleTalk,* second edition.

Specifying and Using a Multinode Address

You use the multinode address format for multinode applications that want to bind several multinode endpoints to the same socket using different node IDs for each. The multinode address format is identical to the DDP address format except that you use a different constant to identify it. See the section

"Specifying a DDP Address" on page 10-5 and the section "The Multinode Address Structure" on page 10-19 for discussion of these fields.

The significant fields for the multinode address format are the network number and node ID. DDP ignores the other fields. You can request specific values for the network number and node ID when binding an endpoint although, as usual, the address returned by the `OTBind` function contains the actual network and node values that the endpoint has been bound to.

DDP delivers any packet addressed to the bound multinode address whether or not a specific socket or DDP type is specified for the destination address of the packet. Applications that have opened multinode endpoints must perform their own filtering if the socket or DDP type values are important.

Registering Your Endpoint's Name

In order for you to make the name of your AppleTalk endpoint visible to other applications on a network, you have to register its name. There are two ways to do this. The easiest way is for you to simply use the `OTBind` function to bind your endpoint with the NBP address format or the combined DDP-NBP address format. If you use the NBP address format, during the binding process Open Tranport registers your endpoint's name and dynamically assigns a physical socket to your endpoint. If you use the combined DDP-NBP address format, you can specify the physical socket you want to bind the endpoint to. The `OTBind` function is discussed in the chapter "Endpoints" in this book.

The other way to register an endpoint's name involves several additional steps: You have to first bind your endpoint, open an NBP mapper provider, use the Open Transport name-registration function, `OTRegisterName`, as a separate step, and then close the NBP mapper provider. You must use this more complex method if you want to register more than one endpoint on the same socket.

In either case, Open Transport uses NBP to associate the endpoint's name with its physical address. Once your endpoint is registered, it is a network-visible entity that other applications can locate.

When you register a name with the `OTRegisterName` function, the function returns a unique identifier for the registered name. If you later want to delete the name, you can use this identifier to delete it with the `OTDeleteNameByID` function. This method is more convenient than the alternative `OTDeleteName` function. The `OTRegisterName`, `OTDeleteName`, and `OTDeleteNameByID` functions are discussed in the chapter "Endpoints" in this book. Table 10-1 provides a summary of the Open Transport name-registration functions.

Table 10-1 Open Transport name-registration functions

Function	Provider	Use
`OTBind`	Endpoint	Registers the specified NBP name when you bind with the NBP address or the combined DDP-NBP address formats
`OTRegisterName`	Mapper	Registers the specified name
`OTDeleteName`	Mapper	Removes a name that was previously registered dynamically
`OTDeleteNameByID`	Mapper	Removes a name that was previously registered dynamically

Looking Up Names and Addresses

To communicate with an endpoint, Open Transport needs its physical address. There are endpoint and mapper functions you can use to obtain this address, two of which allow you to specify the endpoint's NBP name. In these instances, Open Transport performs a name lookup that resolves the NBP name into a physical name that it can use to locate the endpoint you want. Table 10-1 provides a summary of the Open Transport functions that create or return endpoint name and address information.

Table 10-1 Open Transport name and address functions

Function	Provider	Use
`OTGetProtAddress`	Endpoint	Obtains your endpoint's DDP address. For connection-oriented endpoints that are connected to another endpoint, it also obtains the remote endpoint's address.
`OTResolveAddress`	Endpoint	Obtains the DDP address that corresponds to the specified NBP name.
`OTLookUpName`	Mapper	Obtains the address for the specified name or a list of addresses for the specified name pattern. You can also use this function to verify that a specified name is still available on the network and that it is associated with a specified address.
`OTATalkGetInfo`	AppleTalk service	Obtains addressing information about the current environment of an AppleTalk node.

You can improve performance in certain circumstances if you use the endpoint `OTResolveAddress` function instead of the mapper `OTLookUpName` function. Calling `OTResolveAddress` resolves the name into a physical address by using information that is maintained in the current node whereas the `OTLookUpName` function has to go out over the network to look up its information. For example, if you are going to use an NBP address structure repeatedly to specify a remote endpoint in a connectionless or transaction-based service, you can speed up your processing if you first use the `OTResolveAddress` function to resolve the NBP address into a DDP address and then subsequently use only that DDP address to specify the remote endpoint. Otherwise, an NBP lookup could occur on the network for every packet and slow down communications.

When you call the `OTLookUpName` function to obtain the address associated with an NBP name, you can specify a name pattern rather than a complete name by using wildcard operators for the variable parts of the name. Table 10-2 shows the wildcard operators that you can use to specify a name pattern for a name specified as a partial name.

Table 10-2 Wildcard operators

Character	Meaning
=	All possible values. You can use the equal sign (=) alone in the name or type field.
≈	Any or no characters in this position. You can use the double tilde (≈) to obtain matches for name or type fields. For example, "pa≈l" matches "pal," "paul," and "paper ball." You can use only one double tilde in any string. If you use the double tilde alone, it has the same meaning as the equal sign (=). Press Option-X to type the double tilde character (≈) on a Macintosh keyboard.
*	Your local zone. You can leave this blank (preferred method) or use the asterisk (*) to indicate the zone to which this node belongs.

Depending on how you structure the name pattern with wildcards, the `OTLookUpName` function can return a list of names if more than one name matches the specified pattern. For example, if you want to retrieve the names and addresses of all the applications defined with a given type, such as mailboxes, in the same zone as the one in which your process is running, you can set the name field to the equal sign (=), set the type field to "Mailbox," and leave the zone field blank. The `OTLookUpName` function returns the NBP names and DDP addresses of all mailboxes in that zone.

Manipulating an NBP Name

If you need to store or manipulate the name, type, or zone part of an NBP name separately, you need to use an **NBP entity structure,** which is a data structure that Open Transport provides for this purpose. Open Transport also provides several utility functions to transfer data between NBP entities and NBP names.

The NBP entity structure holds an NBP name in the form *name*:*type*@*zone*, with each part containing the maximum 32 characters plus a length byte, for a total possible length of 99 bytes. The NBP entity itself does not contain escape characters, but the NBP entity extraction functions insert a backslash (\) in

front of any backslash, colon (:), or at sign (@) they find in an NBP name so that mapper functions can use a correctly formatted NBP name.

You can initialize an NBP entity and then load it with the name, type, and zone of an NBP name individually (the `OTSetNBPName`, `OTSetNBPType`, and `OTSetNBPZone` functions), or you can load an NBP entity with an entire NBP address at one time (the `OTSetNBPEntityFromAddress` function). Once you have loaded an NBP entity, you can find out how much buffer space it actually uses for the NBP name it holds (the `OTGetNBPEntityLengthAsAddress` function). You can then extract each individual NBP name part one at a time (the `OTExtractNBPName`, `OTExtractNBPType`, and `OTExtractNBPZone` functions), or you can copy the entire NBP entity into an NBP address structure (the `OTSetAddressFromNBPEntity` function).

AppleTalk Addressing Reference

This section describes the constants, data structures, and functions used with AppleTalk protocol addresses.

Constants and Data Types

This section describes the constants and data types used for the address formats that are recognized by AppleTalk endpoints: the DDP address structure, the NBP address structure, the combined DDP-NBP address structure, and the multinode address structure.

Basic Constants

You define the length of AppleTalk addresses and NBP name strings as well as identify wildcards used in NBP names by using the constants defined here. The NBP default zone is also defined here, although if you do not use any zone, Open Transport automatically defaults to * for you.

The constant `kNBPEntityBufferSize` specifies the maximum size of the NBP name buffer, currently defined to be 105 bytes. This permits a NBP name string whose name, type, and zone fields each contain the maximum 32 characters, plus 2 bytes for the separator characters (: and @) and 7 bytes for an optional

pad byte and 6 escape characters, which are indicated by the backslash (\) followed by a colon (:), at sign (@), or another backslash. (See "The NBP Address Structure" on page 10-17 and "The Combined DDP-NBP Address Structure" on page 10-18 for examples of its use.)

```
enum {
    kNBPMaxNameLength           = 32,
    kNBPMaxTypeLength           = 32,
    kNBPMaxZoneLength           = 32,
    kNBPSlushLength             = 9,    /* Extra space for @,:,escape chars */
    kNBPMaxEntityLength         = (kNBPMaxNameLength + kNBPMaxTypeLength +
                                    kNBPMaxZoneLength + 3),
    kNBPEntityBufferSize        = (kNBPMaxNameLength + kNBPMaxTypeLength +
                                    kNBPMaxZoneLength + kNBPSlushLength),
    kNBPWildCard                = 0x3D, /* NBP name and type match anything '=' */
    kNBPImbeddedWildCard        = 0xC5, /* NBP name and type match some '≈' */
    kNBPDefaultZone             = 0x2A, /* NBP default zone '*' */

    kZIPMaxZoneLength           = kNBPMaxZoneLength,

    kDDPAddressLength           = 8,
    kNBPAddressLength           = kNBPEntityBufferSize,
    kAppleTalkAddressLength     = kDDPAddressLength + kNBPEntityBufferSize
};
```

Address Format Constants

You identify each AppleTalk address structure by using a specific constant to indicate which address type you want to use. The permitted constants and their values are listed here:

```
enum {
    AF_ATALK_DDP            = 0x0100,       /* DDP address type */
    AF_ATALK_DDPNBP         = 0x0101,       /* DDPNBP address type */
    AF_ATALK_NBP            = 0x0102,       /* NBP address type */
    AF_ATALK_MNODE          = 0x0103        /* multinode address type */
};
```

The DDP Address Structure

You use the DDP address format, specified by the DDP address structure, to identify the physical socket address for your endpoint. The DDP address structure is defined by the `DDPAddress` data type.

```
struct DDPAddress
{
    OTAddressType       fAddressType;
    UInt16              fNetwork;
    UInt8               fNodeID;
    UInt8               fSocket;
    UInt8               fDDPType;
    UInt8               fPad;
};
```

FIELD DESCRIPTIONS

`fAddressType`
A number that specifies the format of the address. Use the constant `AF_ATALK_DDP`.

`fNetwork` A 16-bit number in the range 0 to 65,534 that specifies the network number. The network number 65,535 (all bits set to 1) is reserved by Apple Computer, Inc. The network number 0 specifies the node's local network.

`fNodeID` An 8-bit number in the range from 0 to 255 that specifies the node ID. A node ID of 255 is accepted by all nodes, permitting the broadcasting of packets to all nodes on the network; a node ID of 0 specifies your own local node and is illegal other than at bind time. For other values, refer to *Inside AppleTalk,* second edition.

`fSocket` An 8-bit number in the range of 1 through 254 that specifies a logical entity on your node. A socket number of 0 at bind time instructs Open Transport to dynamically assign a socket number; a socket number of 4 indicates the echo socket. For other values, refer to *Inside AppleTalk,* second edition.

`fDDPType` A number identifying the DDP type field. Unless you are using the DDP protocol directly, set this field to 0. For additional information see the chapter "Datagram Delivery Protocol (DDP)" in this book and *Inside AppleTalk,* second edition.

`fPad` Reserved. Set to 0.

The NBP Address Structure

You use the NBP address format, specified by the NBP address structure, to identify the NBP name associated with your endpoint. The NBP address structure is defined by the `NBPAddress` data type.

```
struct NBPAddress
{
    OTAddressType       fAddressType
    UInt8               fNBPNameBuffer[kNBPEntityBufferSize];
};
```

FIELD DESCRIPTIONS

`fAddressType`
A number that specifies the format of the address. Use the constant `AF_ATALK_NBP`.

`fNBPNameBuffer`
An 8-bit number that specifies the buffer that holds the NBP name string. The string specifies an endpoint name in the format *name*:*type*@*zone* and is not null terminated. You can precede colons (:), at signs (@), and backslash (\) characters with a backslash if you want to include them as part of the name.

The constant `kNBPEntityBufferSize` specifies the maximum size of the buffer, currently defined to be 105 bytes. This permits a string whose name, type, and zone fields each contain the maximum 32 characters, plus 2 bytes for the separator characters (: and @) and 7 bytes for an optional pad byte and 6 escape characters, which are indicated by the backslash (\) followed by a colon (:), at sign (@), or another backslash.

The Combined DDP-NBP Address Structure

You use the combined DDP-NBP address format, specified by the combined DDP-NBP address structure, to identify the physical socket address and the NBP name associated with your endpoint for use when you bind an endpoint. The combined DDP-NBP address structure is defined by the `DDPNBPAddress` data type.

```
struct DDPNBPAddress
{
    OTAddressType       fAddressType;
    UInt16              fNetwork;
    UInt8               fNode;
    UInt8               fSocket;
    UInt8               fDDPType;
    UInt8               fPad;
    UInt8               fNBPNameBuffer[kNBPEntityBufferSize];
};
```

FIELD DESCRIPTIONS

`fAddressType` A number that specifies the format of the address. Use the constant `AF_ATALK_DDPNBP`.

`fNetwork` A 16-bit number in the range 0 to 65,534 that specifies the network number. The network number 65,535 (all bits set to 1) is reserved by Apple Computer, Inc. The network number 0 specifies the node's local network.

`fNodeID` An 8-bit number in the range from 0 to 255 that specifies the node ID. A node ID of 255 is accepted by all nodes, permitting the broadcasting of packets to all nodes on the network; a node ID of 0 specifies your own local node and is illegal other than at bind time. For other values, refer to *Inside AppleTalk,* second edition.

`fSocket` An 8-bit number in the range of 1 through 254 that specifies a logical entity on your node. A socket number of 0 at bind time instructs Open Transport to dynamically assign a socket number; a socket number of 4 indicates the echo socket. For other values, refer to *Inside AppleTalk,* second edition.

`fDDPType` A number identifying the DDP type field. Unless you are using the DDP protocol directly, set this field to 0. For additional information see the chapter "Datagram Delivery Protocol (DDP)" in this book and *Inside AppleTalk,* second edition.

`fPad` Reserved. Set to 0.

`fNBPNameBuffer`
An 8-bit number that specifies the buffer that holds the NBP name string. The string specifies an endpoint name in the format *name*:*type@zone* and is not null terminated. You can precede colons (:), at signs (@), and backslash (\) characters with a backslash if you want to include them as part of the name.

The constant `kNBPEntityBufferSize` specifies the maximum size of the buffer, currently defined to be 105 bytes. This permits a string whose name, type, and zone fields each contain the maximum 32 characters, plus 2 bytes for the separator characters (: and @) and 7 bytes for an optional pad bytes and 6 escape characters, which are indicated by the backslash (\) followed by a colon (:), at sign (@), or another backslash.

The Multinode Address Structure

You use the multinode address format, specified by the DDP address structure, to identify the physical socket address for a multinode endpoint. The DDP address structure is defined by the `DDPAddress` data type, described in the section "The DDP Address Structure."

```
struct DDPAddress
{
    OTAddressType       fAddressType;
    UInt16              fNetwork;
    UInt8               fNode;
    UInt8               fSocket;
    UInt8               fDDPType;
    UInt8               fPad;
};
```

FIELD DESCRIPTIONS

`fAddressType`	A number that specifies the format of the address. Use the constant `AF_ATALK_MNODE`. This is the only way to distinguish the multinode format from a DDP address.
`fNetwork`	A 16-bit number in the range 0 to 65,534 that specifies the network number. The network number 65,535 (all bits set to 1) is reserved by Apple Computer, Inc. The network number 0 specifies the node's local network.
`fNodeID`	An 8-bit number in the range from 0 to 255 that specifies the node ID. A node ID of 255 is accepted by all nodes, permitting the broadcasting of packets to all nodes on the network; a node ID of 0 specifies your own local node and is illegal other than at bind time. For other values, refer to *Inside AppleTalk,* second edition.
`fSocket`	An 8-bit number in the range of 1 through 254 that specifies a logical entity on your node. A socket number of 0 at bind time instructs Open Transport to dynamically assign a socket number; a socket number of 4 indicates the echo socket. For other values, refer to *Inside AppleTalk,* second edition.
`fDDPType`	A number identifying the DDP type field. Unless you are using the DDP protocol directly, set this field to 0. For additional information see the chapter "Datagram Delivery Protocol (DDP)" in this book and *Inside AppleTalk,* second edition.
`fPad`	Reserved. Set to 0.

The NBP Entity Structure

You use an NBP entity to more conveniently manipulate NBP names because it allows you to extract and set the NBP name's three parts (name, type, and zone) separately. Its use is optional under Open Transport, but it provides an easier way to port programs written for classic AppleTalk. There are many AppleTalk utility functions that transfer data between NBP entity structures and NBP names.

The NBP entity structure is defined by the `NBPEntity` data type.

```
struct NBPEntity
{
    UInt8           fEntity[kNBPMaxEntityLength];
};
```

FIELD DESCRIPTIONS

`fEntity` An 8-bit number that specifies the NBP entity you wish to use to hold the NBP name.

The NBP entity holds an NBP name in the form *name*:*type*@*zone*, and the constant `kNBPMaxEntityLength` specifies the maximum size of the buffer, currently defined to be 99 bytes. This permits an NBP name whose name, type, and zone contain the maximum 32 characters each plus a length byte. The NBP entity itself does not contain escape characters, but the NBP entity extraction functions add them as necessary when converting NBP name strings from NBP entities.

Functions

This section describes AppleTalk utility functions that initialize DDP and NBP data structures, that compare DDP addresses, and that transfer data between NBP entities and NBP names.

OTInitDDPAddress

Initializes a DDP address structure.

C INTERFACE

```
void OTInitDDPAddress(DDPAddress* address, UInt16 net,
                      UInt8 node, UInt8 socket, UInt8 ddpType);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`address`	A pointer to the DDP address structure you wish to initialize.
`net`	The network number you wish to specify. Set to 0 to default to the local network.
`node`	The node ID you wish to specify. Set to 0 to default to the local node.
`socket`	The socket number you wish to specify. Set to 0 to allow Open Transport to assign a socket dynamically when you use this address to bind an endpoint.
`ddpType`	The DDP type you wish to specify. Set to 0 unless you are using DDP.

SEE ALSO

The DDP address structure is described in the section "The DDP Address Structure," beginning on page 10-16.

To initialize an NBP address, use the `OTInitNBPAddress` function (page 10-22). To initialize a combined DDPNBP address, use the `OTInitDDPNBPAddress` function (page 10-23).

See the chapter "Datagram Delivery Protocol" in this book for more information about the DDP type.

OTInitNBPAddress

Initializes an NBP address structure.

C INTERFACE

```
size_t OTInitNBPAddress(NBPAddress* address, const char* name);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`address`	A pointer to the NBP address structure you wish to initialize.
`name`	A pointer to the NBP string you wish to use for the NBP name.

DESCRIPTION

The `OTInitNBPAddress` function can be used to initialize an NBP address structure with the NBP name specified in the `name` parameter, which is assumed to already be in the correct string format. The function returns the size of the NBP address structure, which is the size of the `fAddressType` field plus the length of the string in the `name` parameter.

SEE ALSO

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To initialize a DDP address, use the `OTInitDDPAddress` function (page 10-21).

To initialize a combined DDPNBP address, use the `OTInitDDPNBPAddress` function (page 10-23).

OTInitDDPNBPAddress

Initializes a combined DDP-NBP address structure.

C INTERFACE

```
size_t OTInitDDPNBPAddress(DDPNBPAddress* address,
                           const char* name, UInt16 net, UInt8 node,
                           UInt8 socket, UInt8 ddpType);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`address`	A pointer to the combined DDP-NBP address structure you wish to initialize.
`name`	A pointer to the NBP string you wish to use for the NBP name.
`net`	The network number you wish to specify. Set to 0 to default to the local network.
`node`	The node ID you wish to specify. Set to 0 to default to the local node.
`socket`	The socket number you wish to specify. Set to 0 to allow Open Transport to assign a socket dynamically when you use this address to bind an endpoint.
`ddpType`	The DDP type you wish to specify. Set to 0 unless you are using DDP.

DESCRIPTION

The `OTInitDDPNBPAddress` function initializes a combined DDP-NBP address structure with the data provided in the parameters: NBP name, network number, node ID, socket number, and DDP type. The function returns the total size of the address structure, which is the length of the `name` parameter plus the size of a `DDPAddress` structure.

SEE ALSO

The combined DDP-NBP address structure is described in the section "The Combined DDP-NBP Address Structure," beginning on page 10-18.

To initialize an NBP address, use the `OTInitNBPAddress` function (page 10-22).

To initialize a DDP address, use the `OTInitDDPAddress` function (page 10-21).

See the chapter "Datagram Delivery Protocol" in this book for more information about the DDP type.

OTCompareDDPAddresses

Compares two DDP address structures.

C INTERFACE

```
Boolean OTCompareDDPAddresses(const DDPAddress* addr1,
                              const DDPAddress* addr2);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`addr1`	A pointer to one of the DDP address structures you wish to compare.
`addr2`	A pointer to the second DDP address structure you wish to compare.

DESCRIPTION

The `OTCompareDDPAddresses` function compares two DDP addresses for equality and returns `true` if the two addresses match. It cannot compare NBP or combined DDP-NBP addresses; using these address types always returns `false`. This function uses the zero-matches-anything AppleTalk rule when doing the matching, which means that a value of 0 in any field results in an acceptable match.

OTInitNBPEntity

Initializes an NBP entity structure.

C INTERFACE

```
void OTInitNBPEntity(NBPEntity* nbpEntity);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity` A pointer to the NBP entity structure you wish to initialize.

DESCRIPTION

The `OTInitNBPEntity` function initializes an NBP entity structure, setting the name, type and zone parts of an NBP name to empty strings.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPType` function (page 10-33), the `OTSetNBPType` function (page 10-33), and the `OTSetNBPZone` function (page 10-35), respectively.

OTGetNBPEntityLengthAsAddress

Obtains the size of an NBP entity structure.

C INTERFACE

```
size_t OTGetNBPEntityLengthAsAddress(const NBPEntity* nbpEntity);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure you wish to determine the length of.

SPECIAL CONSIDERATIONS

Use this function to determine the appropriate buffer size for an NBP entity before using the `OTSetAddressFromNBPEntity` function.

DESCRIPTION

The `OTGetNBPEntityLengthAsAddress` function obtains the number of bytes needed to store an NBP entity structure into an NBP or combined DDP-NBP address structure.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

The combined DDP-NBP entity structure is described in the section "The Combined DDP-NBP Address Structure," beginning on page 10-18.

To store an NBP entity structure as an NBP address string, use the `OTSetAddressFromNBPEntity` function (page 10-28).

OTSetAddressFromNBPEntity

Stores an NBP entity structure as an NBP address string.

C INTERFACE

```
size_t OTSetAddressFromNBPEntity(UInt8* nameBuf,
                                 const NBPEntity* nbpEntity);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nameBuf`	A pointer to the NBP address buffer in which you wish to store the NBP entity.
`nbpEntity`	A pointer to the NBP entity you wish to store.

DESCRIPTION

The `OTSetAddressFromNBPEntity` function stores the information in the NBP entity into the buffer specified by the `nameBuf` parameter in the format required for mapper calls—that is, if you have a backslash (\), a colon (:), or an at-sign (@) in your NBP name, this function inserts a backslash before each so that the mapper functions can handle them correctly. This function returns the number of bytes that were actually used in the buffer.

SPECIAL CONSIDERATIONS

Use the `OTGetNBPEntityLengthAsAddress` function beforehand to determine the appropriate buffer size.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To determine the appropriate buffer size for an NBP entity, use the `OTGetNBPEntityLengthAsAddress` function (page 10-27).

To parse and store all or part of an NBP name into an NBP entity, use the `OTSetNBPEntityFromAddress` function (page 10-29).

OTSetNBPEntityFromAddress

Parses and stores an NBP address into an NBP entity.

C INTERFACE

```
Boolean OTSetNBPEntityFromAddress(NBPEntity* nbpEntity,
                                  const UInt8* addrBuf,
                                  size_t len);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity in which you wish to store an address.
`addrBuf`	A pointer to the address buffer in which to store the NBP name string.
`len`	The number of characters to parse and store.

DESCRIPTION

The `OTSetNBPEntityFromAddress` function parses an NBP address or a combined DDP-NBP address into the NBP name's constituent parts (name, type, and zone) and stores the result in an NBP entity. The function ignores the DDP address part of a combined DDP-NBP address. From the NBP entity, each of the constituent parts of the name can be later retrieved or changed.

This function returns `true` if it worked successfully; it returns `false` if it had to truncate any data—that is, if the address had data that was too long in one of the fields, each of which only holds 32 characters of data. When this occurs, Open Transport still stores the data, but in a truncated form.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

The combined DDP-NBP entity structure is described in the section "The Combined DDP-NBP Address Structure," beginning on page 10-18.

To copy the contents of an NBP entity into an NBP address structure, use the `OTSetAddressFromNBPEntity` function (page 10-28).

To determine the appropriate buffer size for an NBP entity, use the `OTGetNBPEntityLengthAsAddress` function (page 10-27).

To store the NBP name in an NBP entity, use the `OTSetNBPName` function (page 10-32); to store the NBP type, use the `OTSetNBPType` function (page 10-33); and to store the NBP zone, use the `OTSetNBPZone` function (page 10-35).

To extract the name portion of an NBP name from an NBP entity, use the `OTExtractNBPName` function (page 10-36); to extract the type portion of an NBP name, use the `OTExtractNBPType` function (page 10-37); and to extract the zone portion, use the `OTExtractNBPZone` function (page 10-38).

OTSetAddressFromNBPString

Copies an NBP name string into an NBP address buffer.

C INTERFACE

```
size_t OTSetAddressFromNBPString(UInt8* addrBuf,
                                 const char* nbpName, SInt32 len);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`addrBuf`	A pointer to the NBP address buffer in which to store the NBP name string.
`nbpName`	A pointer to the NBP name string you wish to copy into the buffer.
`len`	The number of characters to copy.

DESCRIPTION

The `OTSetAddressFromNBPString` function copies the string indicated by the `nbpName` parameter into the buffer indicated by the `addrBuf` parameter. The `len` parameter indicates the number of characters to copy. A value of -1 copies the entire `nbpName` string. The function returns the number of bytes actually copied.

SEE ALSO

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To copy the contents of an NBP entity into an NBP address structure, use the `OTSetAddressFromNBPEntity` function (page 10-28).

OTSetNBPName

Stores the name part of an NBP name into an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPName(NBPEntity* nbpEntity, const char* name);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure in which you wish to store an address.
`name`	A pointer to the name portion of an NBP name string that you wish to store.

DESCRIPTION

The `OTSetNBPName` function stores the NBP name specified by the `name` parameter into the NBP entity structure indicated by the `nbpEntity` parameter, deleting any previous name stored there. This function returns `false` if the `name` parameter is longer than the maximum allowed for a name part of an NBP name (32 characters).

SPECIAL CONSIDERATIONS

When you store all or part of an NBP name in an NBP entity structure, do not include the backslash as an escape character. The NBP entity extraction functions insert a backslash (\) in front of any backslash, colon (:), or at-sign (@) they find in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the type and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPType` function (page 10-33) and the `OTSetNBPZone` function (page 10-35), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 10-36), the `OTExtractNBPType` function (page 10-37), and the `OTExtractNBPZone` function (page 10-38), respectively.

OTSetNBPType

Stores the type part of an NBP name in an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPType(NBPEntity* nbpEntity, const char* type);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure in which you wish to store an address.
`type`	A pointer to the type portion of an NBP name string that you wish to store.

DESCRIPTION

The `OTSetNBPType` function stores the NBP type specified by the `type` parameter into the NBP entity structure indicated by the `nbpEntity` parameter, deleting any previous type stored there. The type supplied must not have any escape characters stored in it, although you do not receive any error message if you do use such characters. This function returns `false` if the `type` parameter is longer than the maximum allowed for type part of an NBP name (32 characters).

SPECIAL CONSIDERATIONS

When you store all or part of an NBP name in an NBP entity structure, do not include the backslash as an escape character. The NBP entity extraction functions insert a backslash (\) in front of any backslash, colon (:), or at-sign (@) they find in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the name and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 10-32) and the `OTSetNBPZone` function (page 10-35), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 10-36), the `OTExtractNBPType` function (page 10-37), and the `OTExtractNBPZone` function (page 10-38), respectively.

OTSetNBPZone

Stores the zone part of an NBP name in an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPZone(NBPEntity* nbpEntity, const char* zone);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure in which you wish to store an address.
`zone`	A pointer to the zone portion of an NBP name string that you wish to store.

DESCRIPTION

The `OTSetNBPZone` function stores the NBP zone specified by the `zone` parameter into the NBP entity structure indicated by the `nbpEntity` parameter, deleting any previous zone stored there. The zone supplied must not have any of the NBP escape characters stored in it, although you do not receive any error message if you do use such characters. This function returns `false` if the `zone` parameter is longer than the maximum allowed for zone part of an NBP name (32 characters).

SPECIAL CONSIDERATIONS

When you store all or part of an NBP name in an NBP entity structure, do not include the backslash as an escape character. The NBP entity extraction functions insert a backslash (\) in front of any backslash, colon (:), or at-sign (@) they find in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the name and type parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 10-32) and the `OTSetNBPType` function (page 10-33), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 10-36), the `OTExtractNBPType` function (page 10-37), and the `OTExtractNBPZone` function (page 10-38), respectively.

OTExtractNBPName

Extracts the name part of an NBP name from an NBP entity structure.

C INTERFACE

```
void OTExtractNBPName(const NBPEntity* nbpEntity, char* name);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure from which you wish to extract an address.
`name`	A pointer to the string buffer in which to store the name portion of an NBP name string that you wish to extract from the NBP entity.

DESCRIPTION

The `OTExtractNBPName` function extracts the name part of an NBP name from the specified NBP entity structure and stores it into the string buffer specified by the `name` parameter. This function inserts a backslash (\) in front of any backslash, colon (:), or at-sign (@) it finds in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 10-32), the `OTSetNBPType` function (page 10-33), and the `OTSetNBPZone` function (page 10-35), respectively.

To extract the type and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPType` function (page 10-37) and the `OTExtractNBPZone` function (page 10-38), respectively.

OTExtractNBPType

Extracts the type part of an NBP name from an NBP entity structure.

C INTERFACE

```
void OTExtractNBPType(const NBPEntity* nbpEntity, char* type);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure from which you wish to extract an address.
`type`	A pointer to the string buffer in which to store the type portion of an NBP name string that you wish to extract from the NBP entity.

DESCRIPTION

The `OTExtractNBPType` function extracts the type part of an NBP name from the specified NBP entity structure and stores it into the string buffer specified by the `type` parameter. This function inserts a backslash (\) in front of any backslash, colon (:), or at-sign (@) it finds in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 10-32), the `OTSetNBPType` function (page 10-33), and the `OTSetNBPZone` function (page 10-35), respectively.

To extract the name and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 10-36) and the `OTExtractNBPZone` function (page 10-38), respectively.

OTExtractNBPZone

Extracts the zone part of an NBP name from an NBP entity structure.

C INTERFACE

```
void OTExtractNBPZone(const NBPEntity* nbpEntity, char* zone);
```

C++ INTERFACES

None. C++ applications use the C interface to this function.

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure from which you wish to extract an address.
`type`	A pointer to the string buffer in which to store the type portion of an NBP name string that you wish to extract from the NBP entity.

DESCRIPTION

The `OTExtractNBPZone` function extracts the zone part of an NBP name from the specified NBP entity structure and stores it into the string buffer specified by the `zone` parameter. This function inserts a backslash (\) in front of any backslash, colon (:), or at-sign (@) it finds in an NBP name so that mapper functions can use a correctly formatted NBP name.

SEE ALSO

The NBP entity structure is described in the section "The NBP Entity Structure," beginning on page 10-20.

The NBP address structure is described in the section "The NBP Address Structure," beginning on page 10-17.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 10-32), the `OTSetNBPType` function (page 10-33), and the `OTSetNBPZone` function (page 10-35), respectively.

To extract the name and type parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 10-36) and the `OTExtractNBPType` function (page 10-37) , respectively.

AppleTalk Service Providers

Contents

The AppleTalk service provider is an Open Transport provider that gives you access to zone and node information functions that are specific to the AppleTalk protocol family. AppleTalk networks use zones to define logical groups of users, and there are several Open Transport functions you can use to determine your endpoint's zone and the zone or zones in your endpoint's network. Open Transport also provides a function that can supply information about your endpoint's AppleTalk environment. To use these functions, you must create a specialized Open Transport provider: an AppleTalk service provider.

The AppleTalk service provider is able to provide information about zones by implementing a subset of the Zone Information Protocol (ZIP), which maps network numbers to zone names for all networks belonging to an AppleTalk internet.

This chapter describes the AppleTalk service provider functions. You should read this chapter if you want to obtain

- the zone name for the node on which your application is running
- the names of the zones for the local network to which your application's node is connected
- the names of all the zones that exist throughout the AppleTalk internet to which your local network belongs
- information about the AppleTalk environment for a given node, including the address of a local router

For an overview of the AppleTalk service provider and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book. Zones are part of the NBP name used in the NBP address format; for more information on this format, read the chapter "AppleTalk Addressing" in this book. For a detailed description of the ZIP specification, see *Inside AppleTalk*, second edition.

About AppleTalk Service Providers

The AppleTalk service provider gives applications access to information and services that are specific to the AppleTalk protocol family. For example, you can obtain zone names and information about the AppleTalk environment for a given machine.

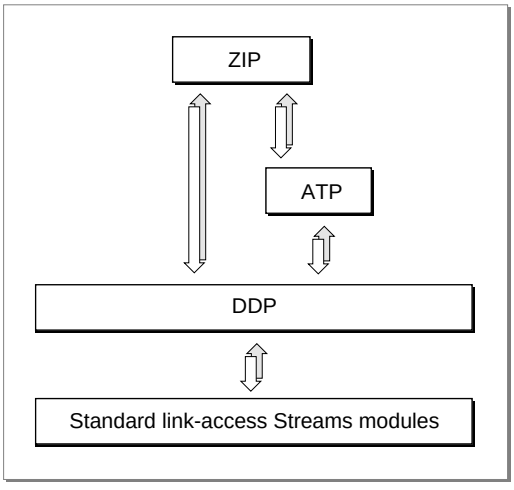
The portion of ZIP implemented by AppleTalk service provider functions can query routers for information about a client's own node, the names of all the zones on the node's local network, or the names of all the zones throughout the AppleTalk internet. AppleTalk routers implement the full set of ZIP functions, with each router maintaining a complete mapping of network numbers and zone names in a zone information table that it periodically updates.

The mapping observes the following rules:

- Every node on a network belongs to only one zone.
- A nonextended LocalTalk network contains only one zone; all nodes in that network belong to that zone.
- A single zone can include nodes that belong to different networks.
- Each AppleTalk extended network has associated with it a list of the zones to which its nodes can belong.

Figure 11-1 shows how, in providing access to the Zone Information Protocol (ZIP), AppleTalk service providers encompass underlying delivery protocols and link-access Stream modules. Because some AppleTalk service provider functions use AppleTalk Transaction Protocol (ATP) packets and some do not, an AppleTalk service provider is considered a client of both ATP and the Datagram Delivery Protocol (DDP).

Figure 11-1 AppleTalk service providers and their underlying delivery mechanism



Using AppleTalk Service Providers

This section explains how you open an AppleTalk service provider and how you use its functions to obtain

- the name of the zone for your application's node
- the names of the zones in your local network or AppleTalk internet
- information about your current AppleTalk environment

You can use AppleTalk service provider functions to get the name of your node's zone. If you are running on a node that belongs to an extended network, you can call an AppleTalk service provider function to get a list of all the zone names associated with that network. For example, an AppleTalk control panel could call the `OTATalkGetLocalZones` function to provide the user with a list of local zones.

You can also use AppleTalk service provider functions in conjunction with mapper functions (described in the chapter "Mappers" in this book). For example, you can use an AppleTalk service provider to look up zones on the

network, then use the mapper function `OTLookUpName` to look up the names in each zone.

Obtaining AppleTalk Service Providers

In order to use the zone and network information functions, you must open an AppleTalk service provider. As with other Open Transport providers, you can open these providers synchronously or asynchronously, and in many ways, they behave similarly to endpoint and mapper providers. For example, you open an AppleTalk service provider by calling either the `OTOpenAppleTalkServices` function or the `OTAsyncOpenAppleTalkServices` function, both of which return an AppleTalk service provider reference to identify the provider you just opened. You use this reference in AppleTalk service provider functions just as you use an endpoint reference in most endpoint provider functions. If you open more than one AppleTalk service provider, the AppleTalk service provider reference lets you to distinguish one provider from another.

If you want to open the AppleTalk service provider asynchronously, you need to create a notifier function that Open Transport can use to send you completion events and other function-specific information. This notifier is the same as the one you need to use for asynchronous endpoints. If you create more than one asynchronous AppleTalk service provider, you can only have one call of each function awaiting completion at the same time.

When you are done using the functions provided by the AppleTalk service provider, you must explicitly close the provider with the generic Open Transport function, `OTCloseProvider`, to release the memory it uses. The `OTCloseProvider` function is described in the chapter “Providers” in this book.

Working With AppleTalk Zones

The NBP name used in the NBP address format has three parts, one of which is the zone name. A **zone** is a logical grouping of nodes within an AppleTalk network. You do not specify the zone when you bind an endpoint; you obtain this value from the system.

An AppleTalk zone name is stored as a Pascal string that contains a maximum of 32 characters. When you add a length byte, you have a string that can have a maximum of 33 bytes. You need to calculate the amount of buffer space you need based on this maximum string size.

The OTATalkGetMyZone function only returns one zone name, so an appropriate buffer size would be 33 bytes. The OTATalkGetLocalZones function, however, returns all the zone names in an extended network, which can hold up to 254 zones, so a maximum buffer size for this function would be 8382 bytes. Because zone names often use less than 32 characters and AppleTalk service providers don't pad short names, 6 KB is likely to be a safe value for this buffer's size.

A much larger buffer would be needed for the OTATalkGetZoneList function, which returns all the zones in all the networks in your AppleTalk internet. You can end up with up to 64 KB of data. To keep the buffer as small and efficient as possible, you can set up a large buffer, test for the kOTBufferOverflowErr error, and then increase the size of the buffer and reissue the call if this error is returned.

Note that these functions, OTATalkGetMyZone, OTATalkGetLocalZones, and OTATalkGetZoneList, return data to you using the TNetbuf structure. This means that you have to define your buffer size in the maxlen field of the TNetbuf structure.

For more information about using zones in NBP names and addresses, see the chapters "Introduction to AppleTalk" and "AppleTalk Addressing" in this book.

Getting the Name of Your Application's Zone

You can get the name of your application's zone by calling the OTATalkGetMyZone function. If you call this function asynchronously, the event T_GETMYZONECOMPLETE signals the completion of the function, and your notifier's cookie parameter points to the zone name with the zone parameter.

Listing 11-1 shows the synchronous application-defined DoGetMyZone function, which opens an AppleTalk service provider and calls the OTATalkGetMyZone function. Note that the length of the buffer, a TNetbuf structure, is set to 0. Open Transport adjusts it to the actual length of the zone name when the function returns. Note also that the function adds a NULL character to the zone name. This is optional, but adding the NULL character turns the string into a C string and makes it easier to handle if you have further use for this string.

Another item to note is that the listing uses the recommended configuration string, the constant kDefaultAppleTalkServicesPath. Open Transport recommends using this string, not the kZIPName constant.

Listing 11-1 Using the DoGetMyZone function synchronously

```
OSStatus DoGetMyZone (char* zoneName)
{
    OSStatus    result;
    ATSvcRef    svcRef;
    TNetbuf     zoneNetbuf;

    svcRef = OTOpenAppleTalkServices
                        (kDefaultAppleTalkServicesPath, 0, &result);
    if (result == noErr)
    {
        zoneNetbuf.maxlen = 33;
        zoneNetbuf.len = 0;
        zoneNetbuf.buf = zoneName;
        OTATalkGetMyZone(svcRef, &zoneNetbuf);
        zoneName[zoneNetBuf.len] = '\0';
        result = OTCloseProvider(svcRef);
    }
    return result;
}
```

Getting a List of Zone Names for Your Local Network or Internet

If you are on an AppleTalk extended network, you can get a list of the names of all the zones in your local network by calling the `OTATalkGetLocalZones` function. If you are on a nonextended network, your network is all on the same zone, and this function returns an asterisk (*), which is the same result as you would get from using the `OTATalkGetMyZone` function.

If you call the `OTATalkGetLocalZones` function asynchronously, the event `T_GETLOCALZONESCOMPLETE` signals the completion of the function, and your notifier's `cookie` parameter points to a list of zone names with the `zones` parameter.

If you are on a network that is part of an AppleTalk internet, you can also use the `OTATalkGetZoneList` function to obtain a list of all the zones in the AppleTalk internet to which your node's network belongs. As with the `OTATalkGetLocalZones` function, if you call the `OTATalkGetZoneList` function asynchronously, Open Transport sends your notifier a completion event, in this

case the T_GETZONELISTCOMPLETE event, to signal the completion of the function, and your notifier's cookie parameter points to a list of zone names with the zones parameter.

It is your responsibility to allocate a buffer that is large enough to hold the list of zone names returned. See the section "Working With AppleTalk Zones" on page 11-6 for more information about buffer sizes.

Getting Information About Your Current AppleTalk Environment

You can use the function OTATalkGetInfo to access an AppleTalk information structure (of type AppleTalkInfo) that contains information about the AppleTalk environment for the node in which your application is running. This information can be useful if you are configuring a network or checking that a network has been configured correctly.

If your application's network is nonextended, this function provides your application's DDP address and the address of the closest local router. If your application's network is extended, this function sets a flag indicating that it's an extended network and provides your application's DDP address, the address of the closest local router, and the current network range for the extended network to which your node belongs.

In either case, this function can also set two other flags: one that indicates that there is a router on the same network, and one that indicates that the network only has one zone.

If you call this function synchronously, the AppleTalk service provider uses the info parameter to provide information about your current network environment. If you call this function asynchronously, the event T_GETATALKINFOCOMPLETE signals the completion of the function, and your notifier's cookie parameter points to the AppleTalk environment information with the info parameter.

If the node is multihoming—that is, if multiple network numbers and node numbers are associated with the same node—the OTATalkGetInfo function returns information about the node whose network number and node ID are selected in the AppleTalk control panel.

AppleTalk Service Provider Reference

This section describes the data structures and functions that are specific to the AppleTalk service provider.

Constants and Data Types

This section describes the events you can receive with the notifier function you provide for your AppleTalk service provider. It also describes the `AppleTalkInfo` data type, which is a structure used by the AppleTalk service provider to return information about your current AppleTalk environment.

To open an AppleTalk service provider, you specify the constant `kDefaultAppleTalkServicesPath` as the `cfig` parameter for the open provider functions, `OTAsyncOpenAppleTalkServices` and `OTOpenAppleTalkServices`.

```
#define kDefaultAppleTalkServicesPath((OTConfiguration*)-3)
```

Completion Event Constants

As with Open Transport endpoint providers, when you call AppleTalk service provider functions asynchronously, Open Transport signals the function's completion by calling the notifier function you installed for your provider and sending it a completion event. The notifier function is the same as the one you used for your endpoint providers, and Open Transport uses its parameters in the same way.

This list gives the completion events that Open Transport returns for each of the AppleTalk service provider functions:

Event constant	Value	Function completed
`T_OPENCOMPLETE`	0x20000007	`OTAsyncOpenAppleTalkServices`
`T_GETMYZONECOMPLETE`	0x23010001	`OTATalkGetMyZone`
`T_GETLOCALZONESCOMPLETE`	0x23010002	`OTATalkGetLocalZones`
`T_GETZONELISTCOMPLETE`	0x23010003	`OTATalkGetZoneList`
`T_GETATALKINFOCOMPLETE`	0x23010004	`OTATalkGetInfo`

The AppleTalk Information Structure

You use the AppleTalk information structure to obtain information about the current AppleTalk environment for the node on which your application is running. The AppleTalk information structure is defined by the `AppleTalkInfo` data type.

```
struct AppleTalkInfo {
    DDPAddress      fOurAddress;
    DDPAddress      fRouterAddress;
    UInt16          fCableRange[2];
    UInt16          fFlags;
};
```

Field descriptions

`fOurAddress` The network number and node ID of your node.

`fRouterAddress` The network number and node ID of the closest router on your network.

`fCableRange` A two-element array indicating the first and last network numbers for the current extended network to which the machine is connected. For nonextended networks, this returns an asterisk (*).

`Flags` A set of flag bits that describe the network:

Flag	Value	Description
`kATalkInfoIsExtended`	`0x0001`	The current network is an extended network.
`kATalkInfoHasRouter`	`0x0002`	There is a router on the same network as this machine.
`kATalkInfoOneZone`	`0x0004`	This network has only one zone.

SEE ALSO

Use the `OTATalkGetInfo` function (page 11-21) to obtain the `AppleTalkInfo` data.

Functions

You use the AppleTalk service provider functions to obtain information about zones and about the network to which your node is connected. Before you can call these functions, you must open the AppleTalk service provider by calling one of the two functions described in the next section.

Opening an AppleTalk Service Provider

Before you can call AppleTalk service provider functions, you must open an AppleTalk service provider by calling the `OTAsyncOpenAppleTalkServices` function or the `OTOpenAppleTalkServices` function.

OTAsyncOpenAppleTalkServices

Opens an asynchronous AppleTalk service provider.

C INTERFACE

```
OSStatus OTAsyncOpenAppleTalkServices(OTConfiguration* cfig,
                                      OTOpenFlags flags,
                                      OTNotifyProcPtr proc,
                                      void* contextPtr);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`cfig` A pointer to a configuration structure that specifies the AppleTalk service provider's characteristics. You can obtain this pointer by using the constant `kDefaultAppleTalkServicesPath` for this parameter. This directs Open Transport to create an AppleTalk service provider on the default hardware port, which is the one selected in the AppleTalk control panel.

`flags`	Reserved. Set to 0.
`proc`	A pointer to your notifier function. Open Transport returns an AppleTalk service provider reference in your notifier's `cookie` parameter.
`contextPtr`	A pointer for your use. The AppleTalk service provider passes this value unchanged to your notifier function.

DESCRIPTION

The `OTAsyncOpenAppleTalkServices` function opens an AppleTalk service provider and gives you a unique AppleTalk service provider reference for it. This function sets the mode of all subsequently called AppleTalk service provider functions as asynchronous.

If you call this function, you must provide a pointer to a notifier function that Open Transport can call to notify you that the function has completed and to return other information that you might need.

When the `OTAsyncOpenAppleTalkServices` function completes, Open Transport calls the notifier function identified in the `proc` parameter. Open Transport returns an AppleTalk service provider reference in your notifier's `cookie` parameter and returns the `T_OPENCOMPLETE` completion event as the event code. The reference value identifies the AppleTalk service provider that you have opened, and you need to supply it as a parameter when you call any AppleTalk service provider function.

▲ **WARNING**

The `OTAsyncOpenAppleTalkServices` function destroys the configuration structure returned by the `OTCreateConfiguration` function. You cannot use the same configuration structure to open multiple AppleTalk service providers. To obtain a valid copy of the configuration structure to use for opening another AppleTalk service provider, use the `OTCloneConfiguration` function. ▲

SPECIAL CONSIDERATIONS

When you no longer need to use AppleTalk service provider functions, you must call the generic Open Transport function `OTCloseProvider`.

COMPLETION EVENT CODES

T_OPENCOMPLETE	0x20000007	The OTAsyncOpenAppleTalkServices function has completed.

SEE ALSO

To open a provider synchronously, use the OTOpenAppleTalkServices function (page 11-14).

You can make a subsequent synchronous request by calling the OTSetSynchronous function, described in the chapter "Providers" in this book.

To create a copy of the configuration structure used to open an AppleTalk service provider, use the OTCloneConfiguration function, described in the chapter "Configuration Management" in this book.

To close the AppleTalk service provider, call the OTCloseProvider function, described in the chapter "Providers" in this book.

OTOpenAppleTalkServices

Opens a synchronous AppleTalk service provider.

C INTERFACE

```
ATSvcRef OTOpenAppleTalkServices(OTConfiguration* cfig,
                                 OTOpenFlags flags,
                                 OSStatus* err);
```

C++ INTERFACE

None. C++ clients use the C interface to this function.

PARAMETERS

`cfig`	A pointer to a configuration structure that specifies the AppleTalk service provider's characteristics. You can obtain this pointer by using the constant `kDefaultAppleTalkServicesPath` for this parameter. This directs Open Transport to create an AppleTalk service provider on the default hardware port, which is the one selected in the AppleTalk control panel.
`flags`	Reserved. Set to 0.
`err`	A pointer to a variable of type `OSStatus` that holds the result code for this function. A value of 0 (`noErr`) indicates successful completion.

DESCRIPTION

The `OTOpenAppleTalkServices` function opens an AppleTalk service provider and gives you a unique AppleTalk service provider reference for it. This function also sets the mode of all subsequently called AppleTalk service provider functions as synchronous.

Because the `OTOpenAppleTalkServices` function operates synchronously, it is recommended that you use the `OTAsyncOpenAppleTalkServices` function instead.

▲ **WARNING**
The `OTOpenAppleTalkServices` function destroys the configuration structure returned by the `OTCreateConfiguration` function. You may not use the same configuration structure to open multiple AppleTalk service providers. To obtain a valid copy of the configuration structure to use for opening another AppleTalk service provider, use the `OTCloneConfiguration` function. ▲

SPECIAL CONSIDERATIONS

When you no longer need to use AppleTalk service provider functions, you must call the generic Open Transport function `OTCloseProvider`.

SEE ALSO

To open a provider asynchronously, use the `OTAsyncOpenAppleTalkServices` function (page 11-12).

To create a copy of the configuration structure used to open an AppleTalk service provider, use the `OTCloneConfiguration` function, described in the chapter "Configuration Management" in this book.

To close the AppleTalk service provider, call the `OTCloseProvider` function, described in the chapter "Providers" in this book.

Obtaining Information About Zones

You use the functions described in this section to obtain the name of one or more zones. You can get the zone name of the node on which your application is running, or if your application is running on a node that belongs to an extended network, you can get the names of all zones in the node's local network or the names of all zones on the AppleTalk internet to which the local network belongs.

OTATalkGetMyZone

Obtains the AppleTalk zone name of the node on which your application is running.

C INTERFACE

```
OSStatus OTATalkGetMyZone(ATSvcRef ref, TNetbuf* zone);
```

C++ INTERFACE

```
TAppleTalkServices::GetMyZone(TNetbuf* zone)
```

PARAMETERS

`ref`	The reference value of your AppleTalk service provider.
`zone`	A pointer to a `TNetbuf` structure that you use to get your application's AppleTalk local zone name.

DESCRIPTION

The `OTATalkGetMyZone` function gets the name of your application's AppleTalk zone. If you call this function asynchronously, Open Transport calls your application's notifier with a `T_GETMYZONECOMPLETE` completion event to signal the function's completion and uses your notifier's `cookie` parameter for the zone name. More precisely, the `cookie` parameter points to a `TNetbuf` structure that in turn points to a buffer containing the zone name, which is stored as a Pascal-style string. The string can be up to 32 characters in length, so with the addition of a length byte, the buffer can have a maximum size of 33 bytes. Using a Pascal-style string for the zone name is redundant since you can determine the length of the string from the `maxlen` field of the `TNetbuf` structure, but the other zone-related calls use Pascal-style strings, so this call also uses them for consistency.

COMPLETION EVENT CODES

`T_GETMYZONECOMPLETE`	0x23010001	The `OTATalkGetMyZone` function has completed.

SEE ALSO

To obtain a list of all zones in your extended network, use the `OTATalkGetLocalZones` function (page 11-18).

To obtain a list of all zones on the AppleTalk internet to which your network belongs, use the `OTATalkGetZoneList` function (page 11-19).

OTATalkGetLocalZones

Obtains a list of the zones available on your network.

C INTERFACE

```
OSStatus OTATalkGetLocalZones(ATSvcRef ref, TNetbuf* zones);
```

C++ INTERFACE

```
TAppleTalkServices::GetLocalZones(TNetbuf* zones);
```

PARAMETERS

`ref`	The reference value of your AppleTalk service provider.
`zones`	A pointer to a `TNetbuf` structure that you use to get a list of the local zone names.

DESCRIPTION

The `OTATalkGetLocalZones` function returns a list of the zone names in your application's network if it is an extended network. These are all the zones to which your node can belong. If your application is in a nonextended network, this function returns only one zone name, the same one returned by the `OTATalkGetMyZone` function.

If you execute this function asynchronously, Open Transport calls your notifier function with a `T_GETLOCALZONESCOMPLETE` completion event to signal the function's completion and uses your notifier's `cookie` parameter for the list of zones. The `cookie` parameter actually holds a pointer to a `TNetbuf` structure, which points to a buffer containing a list of zone names, each of which is stored as a Pascal-style string. Using a Pascal-style string for the zone name is redundant since you can determine the length of the string from the `maxlen` field of the `TNetbuf` structure, but the other zone-related calls use Pascal-style strings, so this call also uses them for consistency.

Each string can be up to 32 characters in length, and if you add a length byte, each can have a maximum size of 33 bytes. As there can be a maximum of 254 zones on an extended network, the maximum size of the buffer is 8382 bytes.

Because zone names are often less than 32 characters long and AppleTalk service providers don't pad short names, 6 KB bytes is likely to be a safe value for the buffer's size, defined by the `TNetbuf->maxlen` field.

COMPLETION EVENT CODES

`T_GETLOCALZONESCOMPLETE`	0x23010002	The `OTATalkGetLocalZones` function has completed.

SEE ALSO

To obtain the zone name for the node your process is running on, use the `OTATalkGetMyZone` function (page 11-16).

To obtain a list of all zones on the AppleTalk internet, use the `OTATalkGetZoneList` function (page 11-19).

OTATalkGetZoneList

Obtains a list of all the zones available on the AppleTalk internet.

C INTERFACE

```
OSStatus OTATalkGetZoneList(ATSvcRef ref, TNetbuf* zones);
```

C++ INTERFACE

```
TAppleTalkServices::GetZoneList(TNetbuf* zones);
```

PARAMETERS

`ref`	The reference value of your AppleTalk service provider.
`zones`	A pointer to a `TNetbuf` structure that you use to get a list of all the zones on your current AppleTalk internet.

DESCRIPTION

The `OTATalkGetZoneList` function returns a list of all the zones on the AppleTalk internet to which your network belongs.

If you execute this function asynchronously, Open Transport calls your notifier function with a `T_GETZONELISTCOMPLETE` completion event to signal the function's completion and uses your notifier's `cookie` parameter for the list of zones. The `cookie` parameter actually holds a pointer to a `TNetbuf` structure, which points to a buffer containing a list of zone names, each of which is a Pascal-style string. Using a Pascal-style string for the zone name is redundant since you can determine the length of the string from the `maxlen` field of the `TNetbuf` structure, but the other zone-related calls use Pascal-style strings, so this call also uses them for consistency.

Each string can be up to 32 characters in length, and if you add a length byte, each can have a maximum size of 33 bytes. As AppleTalk internets can have a number of extended networks, you need to allocate a buffer (using the `TNetbuf->maxlen` field) that holds as much as 64 KB of memory. To keep the buffer size as small and efficient as possible, you can set up a large buffer, test for the `kOTBufferOverflowErr` error, and then increase the size of the buffer and reissue the call if this error is returned.

COMPLETION EVENT CODES

`T_GETZONELISTCOMPLETE`	0x23010003	The `OTATalkGetZoneList` function has completed.

SEE ALSO

To obtain the zone name for the node your process is running on, use the `OTATalkGetMyZone` function (page 11-16).

To obtain a list of all local zones, use the `OTATalkGetLocalZones` function (page 11-18).

Obtaining Information About Your AppleTalk Environment

The `OTATalkGetInfo` function provides information about the current AppleTalk environment for a given node. This information is very important if you configure a network and need to determine that each machine on that network is appropriately incorporated and that traffic on the network is flowing as planned.

OTATalkGetInfo

Obtains information about the AppleTalk environment for a given node.

C INTERFACE

```
OSStatus OTATalkGetInfo(ATSvcRef ref, TNetbuf* info);
```

C++ INTERFACE

```
TAppleTalkServices::GetInfo(TNetbuf* info);
```

PPARAMETERS

`ref`	The reference value of your AppleTalk service provider.
`info`	A pointer to a `TNetbuf` structure that you use to get information about your current AppleTalk environment.

DESCRIPTION

The `OTATalkGetInfo` function returns the information contained in the `AppleTalkInfo` data structure that describes your current AppleTalk environment. This includes your network number and node ID, the network number and node ID of a local router, and the current network range for the extended network to which the machine is connected.

If you execute this function asynchronously, Open Transport calls your notifier with a `T_GETATALKINFOCOMPLETE` completion event to signal the function's completion and uses your notifier's `cookie` parameter for the AppleTalk information. The `cookie` parameter actually holds a pointer to a `TNetbuf` structure, which points in turn to a buffer containing the `AppleTalkInfo` structure. The maximum size of this buffer is 22 bytes.

If the machine is multihomed—that is, if multiple network numbers and node numbers are associated with the same machine—the `OTATalkGetInfo` function returns information about the node whose network number and node ID are selected in the AppleTalk control panel.

COMPLETION EVENT CODES

`T_GETATALKINFOCOMPLETE`	0x23010004	The `OTATalkGetInfo` function has completed.

SEE ALSO

The `AppleTalkInfo` data structure is described in the section "Constants and Data Types" (page 11-10).

Datagram Delivery Protocol (DDP)

Contents

This chapter describes how Open Transport implements the Datagram Delivery Protocol (DDP). It explains how you can use DDP to send and receive data across an AppleTalk internet. DDP is a connectionless transactionless service that you use to transmit data in discrete packets, each carrying its own addressing information. DDP is well suited to applications that do not require reliable delivery of data and that do not want to incur the additional processing associated with setting up and breaking down a connection. Because DDP is connectionless and does not include reliability services, it offers faster performance than do the higher-level protocols that add these services. Applications such as diagnostic tools that retransmit packets at regular intervals to estimate averages or such as games that can tolerate packet loss are good candidates for the use of DDP.

A series of DDP packets transmitted over an AppleTalk internet from one node to another might incur some packet loss, for example, as a result of collisions. If you do not plan on implementing recovery from packet loss in your application, but your application requires it, you can consider using an AppleTalk transport protocol such as the AppleTalk Data Stream Protocol (ADSP) or the AppleTalk Transaction Protocol (ATP). These protocols protect against packet loss and ensure reliability by using positive acknowledgment with mechanisms for retransmitting packets.

This chapter explains how you

- open and bind a DDP endpoint
- send and receive data using DDP
- set checksum options to verify that a packet has not been corrupted during transmission
- use echo packets to measure network performance
- use multinodes

This chapter begins with a description of DDP and the services that it provides under Open Transport. The section "Using Open Transport Functions With DDP" then gives detailed information about how DDP client applications use the endpoint functions that Open Transport provides for connectionless transactionless protocols. For a more detailed explanation of endpoints and their functions, read the chapter "Endpoints" in this book.

For an overview of DDP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter. For more

information about the AppleTalk address formats, see the chapter "AppleTalk Addressing" in this book. For a complete explanation of the DDP specification, see *Inside AppleTalk,* second edition.

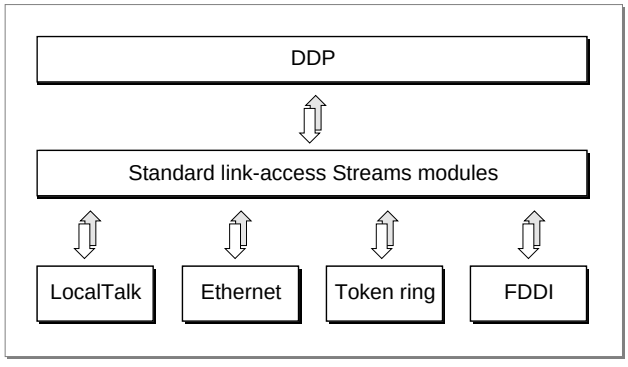
About DDP

The protocol implementations at the physical and data-link layers of the AppleTalk protocol stack provide node-to-node delivery of data on an AppleTalk internet. DDP is a client of the link-access Streams modules, and it extends the node-to-node delivery service provided at the data-link layer by delivering data to a specific socket on a node. A socket number specifies a logical entity on a node and forms part of an AppleTalk endpoint address.

DDP is central to the process of sending and receiving data from endpoint to endpoint across an AppleTalk internet. Regardless of which data link is being used and which (if any) higher-level protocols are providing additional processing, all AppleTalk data is carried in the form of DDP packets, sometimes known as *datagrams*. A packet consists of a header followed by data. DDP delivers data from one endpoint to another by forming the packet header, which contains the destination address, and by passing the packet to the appropriate data link. For packets obtained from the data-link layer, DDP provides what is called a *best-effort delivery service*.

Figure 12-1 shows how the DDP endpoint provider encompasses its underlying link-access Streams modules and its physical ports.

Figure 12-1 The DDP endpoint provider's underlying delivery mechanism



Using DDP

To explicitly use DDP, you open and bind a DDP endpoint. You can then use that endpoint to send or receive data in discrete packets. For outgoing packets, DDP forms the packet header and hands the packet to the appropriate data link. For incoming packets, DDP examines the packet header and attempts to deliver any packet to the specified endpoint as long as the packet meets the following criteria:

- The destination address is valid.
- The default type of the packet matches that of the endpoint to which it is sent.
- The length of the packet matches the length specified in the packet header and does not exceed the maximum for a DDP packet.

If any of these conditions is not satisfied, DDP discards the packet without notifying either the sender or the receiver of the packet. In addition, DDP has no provision for requesting the sender to retransmit a lost or damaged packet.

Binding a DDP Endpoint

As with any endpoint, before you can use it to send or receive data, you must bind it to a physical address. The `OTBind` function takes three parameters: one that specifies the endpoint to be bound, one that requests a specific address, and one that returns the actual address to which Open Transport bound the endpoint.

When binding a DDP endpoint, you can request a particular DDP address, including a static socket address. You can also choose to only specify a DDP type for the endpoint, in which case you set the other fields of the DDP address structure to 0 and allow DDP to dynamically assign a socket. The chapter "AppleTalk Addressing" describes the different address formats you can use to specify an endpoint address.

When you bind a DDP endpoint, there are a few considerations to bear in mind. For example, you do not have to specify the endpoint's socket and the DDP type, but DDP behaves differently depending on whether you specify them or not. Highlighted here are the points to remember:

- If you bind without specifying a socket, DDP uses a dynamically assigned one; if you specify a socket, DDP tries to use it (a statically assigned socket).
- If you bind without specifying a DDP type or if you use a DDP type of 0, Open Transport sets the endpoint's DDP type to a value of 11. This gives you exclusive access to the socket, which means that no other endpoint can bind to it.
- If you bind using a specific DDP type, Open Transport sets the endpoint's DDP type to that value. Any other endpoint that you subsequently bind to that socket must have a unique nonzero DDP type.
- You cannot bind multiple ATP or DDP endpoints on the same socket, although you can bind multiple ADSP or PAP endpoints to the same socket because these endpoints use connection-oriented protocols.
- If you bind with a combined DDP-NBP address, Open Transport uses the DDP part of the address as described in the two preceding bullets. If the bind succeeds, Open Transport registers the NBP name on the endpoint's socket.
- If you bind with an NBP address only, there is no socket number in that form of address, so DDP uses a dynamically assigned socket. If the bind succeeds, DDP registers the endpoint's NBP name on that socket. The endpoint has no default DDP type, so Open Transport sets the DDP type to a value of 11. This has the same effect as described in the earlier bullets.

Using the DDP Type Field to Filter Packet Delivery

You can choose to filter your packet delivery service by using the DDP type field in the endpoint's DDP address structure. The DDP type field is ignored by all protocols other than DDP, so you do not specify the DDP type when passing an address to an AppleTalk endpoint for all protocol layers above DDP.

If you specify a valid nonzero DDP type value when you bind an endpoint, Open Transport uses that value as the default DDP type for that endpoint, using it on all packets sent from that endpoint. If you do not specify a DDP type value or use a value of 0, Open Transport uses a DDP type value of 11 as the default DDP type for that endpoint. If you specify a different DDP type value for any individual packet that you send, Open Transport overrides the endpoint's default DDP type and uses the packet's DDP type.

When receiving incoming packets, a specified DDP type works as a filter: you only receive packets of that one type. If, however, you bind an endpoint without a DDP type or with a DDP type of 0, you receive all incoming packets.

Using the DDP type field when you bind a DDP endpoint has special significance for both sending and receiving packets, as shown in Table 12-1.

Table 12-1 Effects of using the DDP type field

Task	A nonzero DDP type specified at bind	No DDP type or a DDP type of 0 specified at bind
Send	Open Transport uses this DDP type for outgoing packets unless you specify a different DDP type on a per packet basis.	Open Transport uses a DDP type of 11 for outgoing packets unless you specify a different DDP type on a per packet basis.
Receive	You only receive incoming packets for this DDP type.	You receive all incoming packets.

Using the Self-Send and Checksum Options

DDP has two options you can use to control the behavior of DDP endpoints: the `OPT_SELFSEND` and the `OPT_CHECKSUM` options.

You can use the `OPT_SELFSEND` option with DDP to turn self-sending on, which means that when you send a broadcast packet, you receive a copy of it at your

sending endpoint. To turn this on, you set this option with a value of 1. By default this option is turned on.

When you use a DDP endpoint to send data, you can use the `OPT_CHECKSUM` option to enable the calculation of checksums on outgoing packets. In this case, when the packet is received, DDP calculates a checksum for the packet. If the calculated checksum does not match the packet's checksum, DDP assumes the packet has been corrupted and discards the packet without notifying its sender or receiver.

You can specify the `OPT_CHECKSUM` option on every call to `OTSndUData` and control the use of checksums on a per packet basis, or you can use the `OTOptionManagement` function to enable or disable checksums for all outgoing packets. The checksum option `OPT_CHECKSUM` can have one of two values: `T_NO`, which disables checksums, or `T_YES`, which enables it. By default this option is turned off.

For more information about using options, refer to the chapter "Option Management" in this book.

Using Echo Packets

You can use the AppleTalk Echo Protocol (AEP), a client of DDP, to measure the performance of an AppleTalk network or to test for the presence of a given node. Knowing the approximate speed at which an AppleTalk internet delivers packets is helpful in tuning the behavior of an application that uses a higher-level AppleTalk protocol, such as ATP and ADSP.

AEP is implemented in each node as a DDP client process referred to as the **AEP Echoer.** To use the AEP Echoer, you use the `OTSndUData` function to send a packet, called the **echo request packet,** to the target node, and you use the `OTRcvUData` function to receive a packet in response, called the **echo reply packet.**

AEP uses the statically assigned socket number 4, known as the **echoer socket,** to listen for echo packets. Whenever the endpoint associated with this socket receives a packet, AEP examines the packet's DDP type. A value of 4 identifies it as an AEP packet, and AEP then examines the first byte of the packet's data portion. A value of 1 identifies the packet as an echo request packet (sent out from your endpoint), and a value of 2 identifies the packet as an echo reply packet (returned to your endpoint from the remote node).

If the packet is an echo request packet, AEP changes this first byte to a value of 2 (an echo reply packet) before calling DDP to send the packet back to the socket from which it originated.

To test for the presence of a given node, you can iterate through a series of addresses —sending each several packets. If a node exists, AEP sends a packet back; if the node doesn't exist, no packet returns. Be sure to send each node address several packets in case one or more are lost in transmission.

To measure network performance, you need to know the round-trip time of a packet between two nodes on an AppleTalk internet. This is dependent on such factors as the network configuration, the number of routers and bridges that a packet must traverse, and the amount of traffic on the network. As these change, so does the packet transmission time. ATP protocol options let you specify retry-count and interval numbers whose optimum values you can better assess if you know the average round-trip time of a packet on your application's network.

Here are some general guidelines for using the AEP Echoer to measure network performance:

- Use the maximum packet size that you plan on using in your application.
- Accept only echo reply packets from the target node. Set the DDP type field of your endpoint to 4 to filter out all packets except for AEP packets.
- Send more than one packet and calculate the average round-trip time.

  Typically, you should receive an echo reply packet within a few milliseconds on a LAN and within a few seconds on a WAN. If you do not get a response after about 10 seconds, you can assume that DDP dropped or lost your echo request packet, and you can resend the packet.

  The echo reply packet contains the same data that you sent in the echo request packet. If you send multiple packets to determine an average turnaround time and to compensate for the possibility of lost or dropped packets, you should include different data in the data portion of each packet. This allows you to distinguish between replies to different request packets in the event that either some replies are not delivered in the same order that you sent them or that some packets are dropped.
- Bracket the code that sends and receives echo packets with a call to the `Microseconds` function. This function gives much better resolution than the `TickCount` function. The `Microseconds` function is documented in *Inside Macintosh: Operating System Utilities.*

Working With Multinodes

If you are using DDP, you can specify a multinode address for an endpoint. This allows you to bind endpoints to multiple node addresses on the same physical port, which can be useful for testing. Using only one physical machine, you can use multinode addressing to simulate multiple machines.

If a multinode client sends a broadcast or self-send packet, Open Transport makes copies of the packet for the other multinode clients on the same machine internally, thus reducing traffic on the network.

The significant fields for the multinode address format are the network number and node ID. You can request specific values for these address elements when you bind a multinode endpoint and the `OTBind` function returns the actual network and node values for the address to which Open Transport bound the endpoint. Multinode endpoints must use the `DDP_OPT_SRCADDR` option to specify the source DDP address for outgoing packets on a per-packet basis.

The DDP Source Address Option

DDP has one DDP-specific option, `DDP_OPT_SRCADDR`, which sets the source address for outgoing packets. This option is required for multinode endpoints, such as ARA, but can also be used with other types of endpoints.

The option's value must be a DDP address structure using the `AF_ATALK_DDP` address format. The source network number, node number, and source socket are taken from the DDP address. It is an error for these values to be illegal.

This option allows a multinode endpoint to tell Open Transport which of its several sockets actually sent the packet. If no socket is defined, Open Transport uses the default Socket 11.

Using General Open Transport Functions With DDP

This section describes any special considerations you must take into account for Open Transport functions when you use them with the Open Transport DDP implementation. For example, DDP only works with three of the Open Transport send and receive functions, `OTSndUData`, `OTRcvUData`, and `OTRcvUErr` because only these work with protocols that are connectionless and transactionless. You must be familiar with the function descriptions in the chapter "Endpoints" in this book before reading this section.

OTBind

The `OTBind` function associates a local protocol address with the endpoint you specify with the `ref` parameter. You can only bind one DDP or multinode endpoint to a single protocol address.

If you want to use the AEP Echoer, you must bind your endpoint with a socket number of 4 and a DDP Type of 4 to indicate that the endpoint is using the AEP implementation on the node.

OTSndUData

The `OTSndUData` function sends data through connectionless transactionless protocols.

If you want to use the AEP Echoer, you use this function to send echo packets. You need to set the first data byte to a value of 1 to indicate that it is an echo request packet.

OTRcvUData

The `OTRcvUData` function receives data through connectionless transactionless protocols.

If you want to use the AEP Echoer, you use this function to receive echo packets. The first data byte is set to a value of 2 to indicate that it is an echo reply packet.

DDP Reference

This section defines the constant you use to specify the DDP protocol for option management functions

Options

In order to use any option with DDP, you must indicate which protocol the option is intended for. To do this, you use a constant for the DDP protocol in the `level` field of the `TOption` structure when you specify an option.

```
#define ATK_DDP         'DDP '
```

DDP has one DDP-specific option, `DDP_OPT_SRCADDR`, that sets the source address for outgoing packets.

```
#define DDP_OPT_SRCADDR     0x2101      /* DDP source addr override*/
```

A multinode endpoint must use the `DDP_OPT_SRCADDR` option to specify the source address for outgoing packets on a per-packet basis. This option cannot be used with the `OptionManagement` function. The option's value must be a DDP address structure using the `AF_ATALK_DDP` address format. The source network number, node number, and source socket are taken from the DDP address. It is an error for these values to be illegal.

This option is most often used in conjuction with a multinode endpoint, but it can also be used on normal endpoints.

DDP also allows you to use the generic Open Transport options `OPT_SELFSEND` and `OPT_CHECKSUM`, which are described in the chapter "Option Management" in this book.

AppleTalk Data Stream Protocol (ADSP)

Contents

This chapter describes how Open Transport implements the AppleTalk Data Stream Protocol (ADSP). It explains how you can use ADSP to establish a session to exchange a stream of data between two network processes or applications in which both parties have equal control over the communication. ADSP offers a connection-oriented transactionless service that is particularly well suited to the transfer of large amounts of data.

You should read this chapter if you want to write an application that uses ADSP to exchange a stream of data between two equal parties who can each send and receive data. This chapter explains how you

- create an endpoint that listens passively for incoming connection requests
- send and receive data via ADSP
- divide an ADSP data stream into discrete logical units
- use expedited attention messages with ADSP

This chapter begins with a description of ADSP and the services that it provides under Open Transport. The section "Using Open Transport Functions With ADSP" then gives detailed information about how ADSP client applications use the endpoint functions that Open Transport provides for connection-oriented transactionless protocols. For a more detailed explanation of endpoints and their functions, read the chapter "Endpoints" in this book.

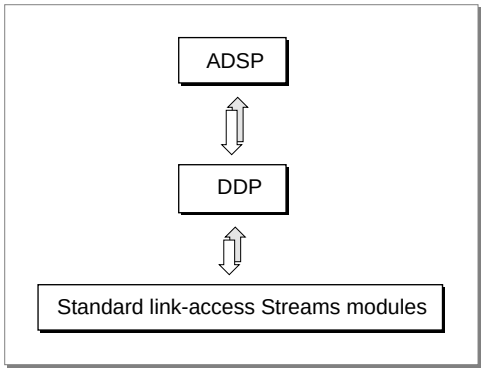
For an overview of ADSP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter. ADSP under Open Transport conforms to the detailed specifications in *Inside AppleTalk,* second edition. See that book for further information about the features mentioned here.

About ADSP

The AppleTalk Data Stream Protocol (ADSP) includes both session and transport services and is the most commonly used of the AppleTalk transport protocols. ADSP allows you to establish and maintain a connection between two AppleTalk network entities and transfer data as a continuous stream. The two clients at either end of an ADSP connection are equal and can perform the same operations.

ADSP, like all other high-level AppleTalk protocols, is a client of the Datagram Delivery Protocol (DDP), which transmits data in packets. However, ADSP builds a session connection on top of DDP's packet transfer services, so you can exchange data as a continuous stream. This allows ADSP's client applications to retain many of the advantages of a transaction-based protocol while providing a full-duplex data stream. Figure 13-1 shows how the ADSP endpoint provider encompasses its underlying delivery protocol and link-access Streams modules.

Figure 13-1 The ADSP endpoint provider's underlying delivery mechanism



Communication between two client applications using ADSP occurs over a connection between two endpoints that provides reliable data delivery. When you bind an ADSP endpoint, the binding process associates a local protocol address with your endpoint. In ADSP, this identifies the socket address, and ADSP uses this as part of the address for sending and receiving packets of data. Each socket can maintain concurrent ADSP connections with several other sockets, but there can be only one ADSP connection between any two sockets at one time.

ADSP uses several internally maintained variables to track its progress as it transmits a data stream across a connection. For example, ADSP associates an internal sequence number with each byte that it sends. This allows ADSP to determine out-of-sequence or duplicate data. ADSP uses the sequence numbers to ensure that the other endpoint receives all of its intended data. If any data does not arrive, ADSP can retransmit it.

The data is available for retransmission because when an endpoint provider sends data to a remote connection end, ADSP first stores it in a buffer, called the **send queue,** and holds the data there until the remote connection end acknowledges receipt. Likewise, when data arrives from a remote endpoint, ADSP stores it in a receiving buffer, called the **receive queue,** until the local endpoint provider acknowledges reading it.

ADSP does not transmit data from the remote connection end until there is space available in the local receive queue. This built-in flow control keeps a connection from being jammed with too much data.

Using ADSP

To use Open Transport ADSP, you first open an endpoint as an ADSP endpoint, which causes Open Transport to allocate the memory ADSP needs for data buffers and for storing the variables ADSP uses to maintain the connection between endpoints. After a connection is established, ADSP manages and controls the data flow between two endpoints throughout a session to ensure that data is delivered and received in the order in which it was sent and that duplicate data is not sent.

As with other connection-oriented protocols, Open Transport ADSP allows you to create a passive endpoint that listens for incoming connection requests rather than initiating such requests. In addition, the implementation of ADSP under Open Transport includes some features that are specific to the two AppleTalk connection-oriented protocols, ADSP and PAP. These are

- an option to enable end-of-message (EOM) indicators that let you break streams of data into logical units
- locally implemented orderly disconnects rather than over-the-wire remote disconnects

A feature unique to ADSP is its separate data channel for expedited data that provides an attention-message facility that lets ADSP endpoints signal each other outside the normal exchange of data.

Binding ADSP Endpoints

You have two choices when you bind an ADSP endpoint: You can create an endpoint that can initiate connections and accept connection requests, or you can create an endpoint that can only receive connection requests.

If the endpoint can initiate connections, you can bind it as a normal Open Transport endpoint and use any of the three AppleTalk address formats for the socket address: DDP, NBP, or the combined DDP-NBP format. If the bind is successful, the endpoint is ready for use in establishing and using a connection.

The other choice when binding an ADSP endpoint is to establish it as a **passive peer** that listens for incoming connection requests. The passive endpoint can accept or deny a connection request based on criteria that you define. The use of a passive peer is typical of a server environment in which a server, such as a file server, is registered with a single name. Endpoints throughout the network can contact the server's passive peer with connection requests. The server can accept or deny a request. It might deny a request, for example, when its resources are exhausted.

To create a passive peer that listens, you specify a queue length greater than 0 during the binding process. The number you use determines how many connection requests the endpoint can support. Once you bind a passive peer, it starts listening for incoming connection requests. When a request arrives, the endpoint retrieves certain information about the request and continues to process it by accepting or rejecting it.

You can bind multiple ADSP endpoints to the same socket, and ADSP can support as many connections on a socket as you have memory for, but you can only have one passive peer that listens on a given socket.

Sending and Receiving ADSP Data

ADSP supports two separate data channels: one for normal data and one for expedited data. You can send a stream of normal data that has no logical boundaries within it that need to be preserved across the connection, or you can use **transport service data units (TSDUs)** to separate the data stream into discrete logical units when sending and receiving data across a connection. For expedited data, you can use **expedited transport service data units,** or ETSDUs.

By default, ADSP does not support TSDUs. Instead, ADSP sends and receives a continuous stream of data with no message delimiters, which means you can exchange data with an endpoint whose protocol does not support TSDUs. If

you do not specify any ADSP-specific options, your packets are not restricted to Open Transport ADSP endpoints, and you can provide transport-independent data transmission.

Open Transport uses a flag in the send and receive functions to indicate multiple sends and receives. The use of this flag, the T_MORE flag, allows you to break up a large data stream without losing its logical boundaries at the other end of the connection. The flag, however, indicates nothing about how the data is packaged for transport on the lower-level protocols below the ADSP endpoint provider.

If you do not use any options, when you receive data, you need to set the T_MORE flag each time you call the OTRcv function. This is because, with a continuous stream of data possible between endpoints, the receiving endpoint is always expecting more data. When you send data, however, you do not use the T_MORE flag.

The Enable EOM (End-of-Message) Option

If transport independence is not crucial for your application, you can use the ADSP enable EOM (OPT_ENABLEEOM) option that allows infinite length TSDUs on the normal data channel.

To send a data stream that is too long for a single TSDU, you set the T_MORE flag on each send to indicate to the remote connection end that another packet is coming that is part of this same message. When a packet arrives without the T_MORE flag set, the remote end knows this is the end of the message. It is possible for this last packet to contain no data because ADSP supports the sending of zero-length packets. This could occur when you send a packet with the T_MORE flag set only to discover that you have no more data to send. In this case, ADSP still expects another packet, but you have no data to put into it. You can send a zero-length packet to set the T_MORE flag correctly.

When you use TSDUs with ADSP, you cannot change the size of the TSDU after you have established the connection with another endpoint. This means that you don't need to double-check the TSDU size after the first packet because it will always be the same for all packets using this connection.

You can enable the EOM option for an endpoint in several ways. One way is to define the option as part of the configuration string you use to open the endpoint. The following line of code enables the EOM option for an ADSP endpoint:

```
OTOpenEndpoint(OTCreateConfiguration("adsp(EnableEOM=1)"),0, NULL, &err);
```

Listing 13-1 shows you another way of setting the enable EOM option. The sample function uses an endpoint reference, `ep`, to identify the endpoint for which you are setting this option and defines a Boolean parameter for enabling or disabling the EOM option. The first task is to define a 4 byte buffer, which is the standard size for options. Then fields and pointers are defined, including the option's level, name, and length. The option's value is set to the Boolean `enableEOM`, which enables the EOM option. Before attempting to negotiate the option, the code makes sure that the endpoint is in synchronous mode. If the endpoint was originally asynchronous, the code restores it to an asynchronous mode before exiting.

Listing 13-1 Setting the enable EOM option

```
OSStatus DoNegotiateEOMOption(EndpointRef ep, Boolean enableEOM){

UInt8           buf[kOTFourByteOptionSize];     /* define 4 byte option buffer */
TOption*        opt;                            /* ptr makes items easier to access */
TOptMgmt        req;
OSStatus        err;
Boolean         wasAsync = false;

opt                 = (TOption*)buf;            /* set option ptr to buffer */
req.opt.buf         = buf;
req.opt.len         = sizeof(buf);
req.flags           = T_NEGOTIATE;              /* negotiate for EOM option */

opt->level          = ATK_ADSP;                 /* it's an ADSP option */
opt->name           = OPT_ENABLEEOM;
opt->len            = kOTFourByteOptionSize;
opt->value[0]       = enableEOM;                /* enable (true) or disable option */

if (!OTIsSynchronous(ep))                       /* check whether ep is sync */
    {
    wasAsync = true;                            /* set flag if async */
    OTSetSynchronous(ep);                       /* set endpoint to sync */
    }
err = OTOptionManagement(ep, &req, &req);
if (wasAsync)                                   /* restore ep state if necessary */
    OTSetAsynchronous(ep);
return err;
}
```

The Checksum Option

You can use the `OPT_CHECKSUM` option to force ADSP to send all outgoing packets with the checksum option enabled. By default, outgoing ADSP packets do not use this option, which directs DDP to compute a checksum and include it in each packet that it sends to the remote endpoint provider. Using checksums slows communications slightly. Normally, ADSP and DDP perform enough error checking to ensure safe delivery of all data, so set this option only if the network is highly unreliable.

Sending Expedited Data

In addition to the full-duplex data stream that an ADSP session maintains, ADSP allows either end of a connection to send an expedited attention message to the other end without interrupting the primary flow of data. Processing expedited data takes precedence over handling normal data, so when an expedited data packet arrives at an endpoint, the endpoint reads this packet before reading the next normal data packet. Both the send and receive functions have a flag, `T_EXPEDITED`, that indicates when a packet has expedited data.

ETSDUs can be up to 572 bytes long, including a 2-byte attention code at the beginning of the user data portion. The minimum ETSDU for ADSP is 2 bytes, so if you send less than that, the data is padded to 2 bytes before being transmitted. If you do not use the attention code, all 572 bytes are available for user data. If you use an attention code, you are responsible for ensuring that the code has a value from $0000 to $EFFF and is not in the reserved range of $F000 to $FFFF.

Note
You cannot use the `OPT_ENABLEEOM` option with the expedited channel or you receive an error message. ◆

Note that not every connection-oriented transactionless protocol supports attention messages or expedited data. Therefore, using this option compromises the transport independence of your application.

Disconnecting

As with all connection-oriented Open Transport protocols, ADSP supports abortive disconnects. In addition, ADSP supports orderly disconnects, although it can only implement them locally.

An abortive disconnect directs the remote endpoint to abruptly tear down its connection without making any accomodation for the data that may be in the transmission pipeline at the time. You can define your own handshake, perhaps using the expedited data channel, to prevent losing data during the disconnection process.

ADSP implements orderly disconnects locally, not over the wire. This means that immediately after you request the disconnect, ADSP sends all data buffered at the local end and then tears down the connection, breaking communication with the remote end. As a result, no data can be sent from either the local or remote endpoint. The endpoints can continue to process data already in their receive queues, but no new data can go out.

Using General Open Transport Functions With ADSP

This section describes any special considerations you must take into account for Open Transport functions when you use them with the Open Transport ADSP implementation. For example, ADSP only works with two of the Open Transport send and receive functions, `OTSnd` and `OTRcv`, because only these work with protocols that are connection-oriented and transactionless. You must be familiar with the function descriptions in the chapter "Endpoints" in this book before reading this section.

OTBind

The `OTBind` function associates a local protocol address with the endpoint provider specified by the `ref` parameter.

You can bind multiple ADSP endpoints to a single protocol address, but you can bind only one passive peer endpoint that listens on that socket.

With ADSP, as with other connection-oriented protocols, the `req->qlen` parameter specifies the number of outstanding connection requests that an endpoint can support. The endpoint can negotiate the final value of `qlen` if it cannot handle the requested number of outstanding connection requests, but in ADSP, the value of `qlen` cannot be negotiated to 0 from a requested value greater than 0.

OTConnect

The `OTConnect` function requests a connection to a specified remote endpoint.

ADSP does not allow application-specific data to be included when you establish a connection, so you need to set the `sndcall->udata.len` field to 0. ADSP ignores the `sndcall->udata.buf` field.

OTRcvConnect

The `OTRcvConnect` function reads the status of a previously issued connection request.

Because ADSP does not allow application-specific data to be associated with a connection request, you need to set the `call->udata.len` field to 0. ADSP ignores the `call->udata.buf` field.

OTListen

The `OTListen` function listens for an incoming connection request.

ADSP does not allow application-specific data to be included when you request a connection, so you need to set the `call->udata.len` field to 0. ADSP ignores the `call->udata.buf` field.

OTAccept

The `OTAccept` function accepts a connection request. You can accept a connection either on the same endpoint that received the connection request or on a different endpoint.

ADSP does not allow application-specific data to be included when you accept a connection, so you need to set the `call->udata.len` field to 0. ADSP ignores the `call->udata.buf` field.

OTSnd

The `OTSnd` function sends normal and expedited data through a connection-oriented transactionless endpoint.

ADSP supports TSDUs through the `OPT_ENABLEEOM` option, although its use compromises the transport independence of your application. In ADSP, TSDUs can be of infinite length and ETSDUs can be up to 572 bytes long. Zero-length packets are supported in ADSP.

OTRcv

The `OTRcv` function receives normal and expedited data through a connection-oriented transactionless endpoint.

ADSP supports TSDUs through the `OPT_ENABLEEOM` option, although its use compromises the transport independence of your application. In ADSP, TSDUs can be of infinite length and ETSDUs can be up to 572 bytes long. Zero-length packets are supported in ADSP.

OTSndDisconnect

The `OTSndDisconnect` function initiates an abortive disconnect or rejects a connection request.

When you call this function with ADSP, you receive a `T_ORDREL` asynchronous event rather than a `T_DISCONNECT` asynchronous event so that you can continue to read in the rest of the data in your receive queue. Otherwise, with a `T_DISCONNECT` event, any remaining unread data is discarded.

In an abortive disconnect, the `call` parameter is ignored because ADSP does not allow application-specific data to be associated with a disconnect. You need to set the `call->udata.len` field to 0. ADSP ignores the `call->udata.buf` field.

OTRcvDisconnect

The `OTRcvDisconnect` function returns information about why a connection attempt failed or an established connection was terminated.

Because ADSP does not allow application-specific data to be associated with a disconnect, you need to set the `discon->udata.len` field to 0. ADSP ignores the `discon->udata.buf` field.

ADSP Reference

This section defines the constant you use to specify the ADSP protocol for option management functions and indicates the generic Open Transport options you can use with ADSP.

Options

In order to use any option with ADSP, you must indicate which protocol the option is intended for. To do this, you use a constant for the ADSP protocol in the `level` field of the `TOption` structure when you specify an option.

```
#define ATK_ADSP        'ADSP '
```

ADSP also allows you to use the `OPT_ENABLEEOM` and the `OPT_CHECKSUM` options, which are described in the chapter "Option Management" in this book.

AppleTalk Transaction Protocol (ATP)

Contents

This chapter describes how Open Transport implements the AppleTalk Transaction Protocol (ATP). It explains how you can use ATP to send requests and responses between ATP endpoints, with one endpoint initiating the request and the other responding to it. You can create an endpoint that can both initiate or respond, or you can create one endpoint that only makes requests and another that only makes responses. Because ATP provides a connectionless transaction-based service, you do not incur the overhead entailed in establishing, maintaining, and breaking a connection that is associated with connection-oriented protocols, such as ADSP, but you can transfer only a limited amount of data using ATP.

You should read this chapter if you want to write an application that requires reliable delivery of data but does not require the transfer of a large amount of data. This chapter explains how you

- open and bind an ATP endpoint
- get information about an ATP endpoint
- use Open Transport functions to initiate and respond to a transaction
- specify ATP options to control connectionless transaction-based services

This chapter begins with a description of ATP and the services that it provides under Open Transport. The section "Using Open Transport Functions With ATP" then gives detailed information about how ATP client applications use the endpoint functions that Open Transport provides for connectionless transaction-based protocols. For a more detailed explanation of endpoints and their functions, read the chapter "Endpoints" in this book.

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

About ATP

The AppleTalk Transaction Protocol (ATP) offers a simple, efficient means of transferring small amounts of data across a network. Using this protocol, one endpoint requests information from another endpoint that possesses the ability to respond to the request. This means that ATP is well-suited to a client-server relation like that used by the Printer Access Protocol (PAP), which uses ATP packets to transport data to printers and allows the printer server to reply with messages to the client workstation that is attempting to print.

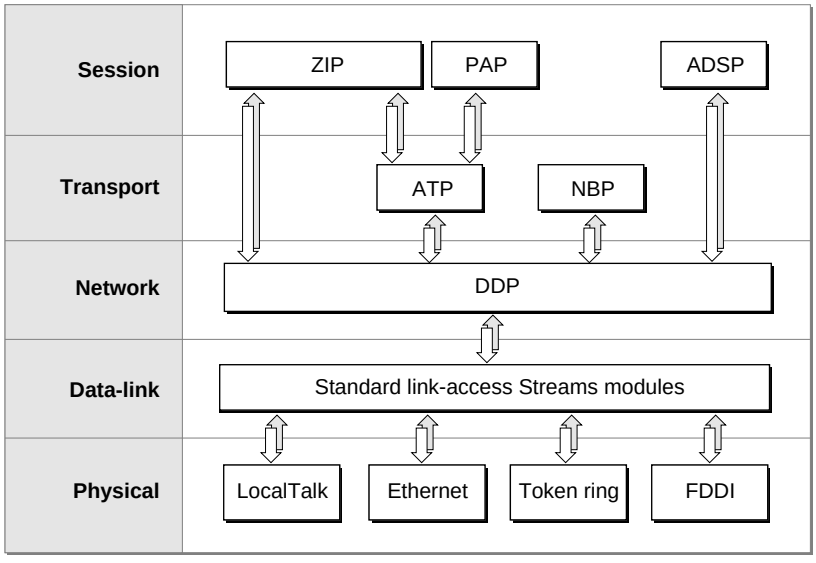
ATP is based on the concept of a transaction. In a **transaction,** one endpoint, called the **requester,** makes a request of another endpoint, called the **responder,** to perform a service and return a response.

You can implement ATP client applications in the following two ways:

- You can write a single application that handles both the requester and responder actions of an ATP transaction and run that application on two networked nodes. This method allows each application to act as either the requester or the responder. However, while each side has the capacity to initiate a transaction, only one side can control the communication during a single transaction.
- You can write two applications, one application that implements the requester part of a transaction and another application that implements the responder side. This model lends itself well to a client-server relation such as PAP, in which many nodes on a network run the requester application (client), while one or more nodes run the responder application (server). One server can respond to transaction requests from various clients.

ATP is a direct client of DDP, and it adds reliable delivery of data to the transport delivery services that DDP provides. ATP ensures that data is delivered without error or packet loss. Figure 14-1 shows how the ATP endpoint provider encompasses its underlying delivery protocol and link-access Streams modules.

Figure 14-1 The ATP endpoint provider's underlying delivery mechanism



Using ATP

In order for two applications to use ATP, each application must have opened and bound an ATP endpoint. The requester initiates a transaction by making a request. When the responder receives the request, it accepts the request, formulates a response that includes any data required by the requester, and sends that response to the requester. When the requester receives the response, the transaction is complete. You can define how often ATP is to retry each request and how long it is to wait between each retry attempt by using the retry count and interval options, described in the section "Specifying ATP Options" on page 14-7.

At-Least-Once and Exactly-Once Transactions

In the course of a transmission, a request might be lost, a response might be lost or delayed, or the responder might fail to acknowledge or accept a request. In any of these situations, the transaction cannot complete. To complete the transaction and assure reliable delivery of data, ATP is responsible for waiting a predetermined amount of time and then retrying the request until it is able to conclude the transaction. If it cannot conclude the transaction, ATP must let the requester know that the attempt has failed. In order to perform these services, ATP supports two types of transactions: at-least-once transactions and exactly-once transactions.

- An **at-least-once transaction** ensures that the responder receives every request directed to it at least once, but this does not prevent the responder from receiving a request more than once. These are also referred to as *ALO transactions*.
- An **exactly-once transaction** ensures that the responder receives a specific request only once. These are also referred to as *XO transactions*. PAP uses this type of ATP transaction.

By default, Open Transport ATP provides XO transaction support for a request transaction by setting the `T_ACKNOWLEDGED` bit in the option flags for the `OTSndURequest` function. This kind of support is appropriate in those cases where harm could be done if a request is satisfied multiple times. For example, if you are identifying a relative memory location that is the result of a calculation, doing the calculation twice would move you to a different location in memory.

In those cases where no harm is done if a request is satisfied multiple times (for example, when the requester asks the responding node to identify itself) you can select ALO transactions by clearing the `T_ACKNOWLEDGED` bit in the option flags for the `OTSndURequest` function.

Sending and Receiving ATP Data

Typically, a requester sends a small amount of data requesting the remote endpoint to take some action or to send back data in reply. The amount of data that the responder can reply with can be quite large. A requester can send only a single ATP packet of 578 bytes, but a responder can return up to eight packets of 578 bytes each, totalling a maximum of 4624 bytes. ATP does not support zero-length packets.

To accomodate the restrictions that a particular network may place on sending that much data at a time, ATP uses the T_MORE flag to communicate to the awaiting requester endpoint when all of the reply data has been accumulated. A single reply may have up to eight packets, and each packet in the reply except for the very last has the T_MORE flag set. The reply data is held at the receiving requester endpoint until a packet arrives that does not have the T_MORE flag set. When this happens, ATP knows that all the reply data has arrived, and it releases the entire reply to the awaiting requester endpoint.

Specifying ATP Options

You can use several options with ATP, but note that doing so compromises the transport independence of your application. There are several ATP-specific options and you can use the generic Open Transport options, OPT_INTERVAL and OPT_RETRYCNT. Table 14-1 summarizes their definitions and default values. All of these options, except ATP_OPT_TRANID, can be set both globally with the OptionManagement function and locally by setting option flags for an individual packet. The ATP_OPT_TRANID option can only be set globally.

Table 14-1 ATP option definitions and default values

Option	Default	Description
OPT_RETRYCNT	8 retries	Sets the number of times ATP retries a request before returning an error to the client.
OPT_INTERVAL	2 seconds	Sets the interval for ATP to wait between retries.
ATP_OPT_RELTIMER	30 seconds	Sets the amount of time the responder must wait for a transaction release packet before it purges a request entry from its transactions list. Acceptable values are 0 (30 seconds), 1 (1 minute), 2 (2 minutes), 3 (4 minutes), 4 (8 minutes).
ATP_OPT_REPLYCNT	8 replies	Specifies the number of replies (1–8) to expect in reply to a request.
ATP_OPT_DATALEN	578 bytes	Sets maximum individual packet size.
ATP_OPT_TRANID	true	Requests a transaction ID.

The Retry Count and Interval Options

After transmitting a transactions from the requester, ATP waits for the interval of time specified by the requester's defined `OPT_INTERVAL` option (default is 2 seconds). If the requester still hasn't received a response from the responder, it retransmits the request. It repeats this process for the number of times defined by the requester's `OPT_RETRYCNT` option (default is 8 retries). Once these maximums have been reached without any response, ATP informs the requester that the responder is unavailable.

The Release Timer Option

With ALO transactions, a responder can receive duplicate requests; with XO transactions, ATP uses additional processing to ensure that a responder receives a request only once. To handle XO transactions safely, the responder maintains a transactions list of all recently received requests. When it receives a request, the responder searches through this list to determine whether it is a new request or a duplicate request. If the request is new, the responder inserts it in the transactions list, time stamping the entry with its time of insertion. If it is a duplicate request and a response has gone out, ATP automatically retransmits the reply without the intervention of the responder application. If it is a duplicate request and a reply has not yet been sent out, ATP discards the request.

When a requester receives a reply from the responder, it sends a transaction release packet to the responder to signal that the transaction has successfully completed, and the responder can now release the transaction from its transactions list. If this transaction release packet is lost, however, the responder would never be able to release the transaction from its list. Because the responder time stamped each new request when it inserted the request into its transactions list, the responder can check the list periodically and eliminate all requests that are older than the time defined by the `ATP_OPT_RELTIMER` option (default is 30 seconds), assuming that these requests remain in the list because the transaction release packet has been lost.

Other ATP-Specific Options

When a reply starts to arrive, the requester needs to know how many packets are in a given reply so that it knows when to stop waiting for more packets. The `ATP_OPT_REPLYCNT` option allows you to define a number between 0 and 8 (the default is 8 packets). You can set this globally, with the `OptionManagement` function, or locally for an individual request.

The `ATP_OPT_DATALEN` option allows you to set the maximum length of an individual packet up to a length of 578 bytes (the default). In most cases, you can leave this at the default. PAP servers, which use a maximum packet size of 512 bytes, can use this option to restrict the ATP packet size. You can set this globally, with the `OptionManagement` function, or locally for an individual request.

The `ATP_OPT_TRANID` option is a Boolean value that, when set to `true`, requests Open Transport to add an option to every request that contains the ATP transaction ID. You can only set this option globally, with the `OptionManagement` function; you cannot set it locally.

Using the ATP Packet Header User Bytes

The first 4 bytes of the ATP packet header contain information that allows Open Transport to identify whether an ATP packet is a request or a response, to specify the sequential position of a response packet, and to identify the transaction. The second 4 bytes of the header are called *user bytes*, and are available for your use. Your application could use the user bytes, for example, to create a simple header for a higher-level protocol.

ATP takes the first 4 bytes of data that the requester specifies and places them in the user bytes portion of the outgoing request. If you do not specify at least 4 bytes of data in the request, ATP pads the user bytes with zeros.

On the responder side, ATP takes the data in the first reply packet's user bytes and puts them into the first 4 bytes of the reply packet's data. ATP ignores the user bytes in all reply packets except for the first packet.

For more information on ATP packets and their header field definitions, refer to *Inside AppleTalk*, second edition.

Using General Open Transport Functions with ATP

This section describes any special considerations you must take into account for Open Transport functions when you use them with the Open Transport ATP implementation. For example, ATP only works with those Open Transport send and receive functions that handle request-reply interactions for connectionless transaction-based protocols. These are `OTSndURequest`, `OTRcvURequest`, `OTSndUReply`, and `OTRcvUReply`. You must be familiar with the descriptions of these functions in the chapter "Endpoints" in this book before reading this section.

OTSndURequest

A client of a connectionless transaction-based protocol such as ATP can use the `OTSndURequest` function to send an ATP request packet to an ATP responder endpoint.

To indicate XO transactions, set the `T_ACKNOWLEDGED` bit in the `OTSndURequest` function's `reqFlags` parameter. To indicate ALO transactions, clear this bit. ATP request packets can have up to 578 bytes, and zero-length TSDUs are not supported.

OTRcvURequest

A client of a connectionless transaction-based protocol such as ATP can use the `OTRcvURequest` function to receive an incoming ATP request packet from an ATP requester endpoint.

On XO transaction packets, the `T_ACKNOWLEDGED` bit in the `OTRcvURequest` function's `reqFlags` parameter is set. On ALO transactions, this bit is clear. ATP request packets can have up to 578 bytes, and zero-length TSDUs are not supported.

OTSndUReply

A client of a connectionless transaction-based protocol such as ATP can use the `OTSndUReply` function to send an ATP reply packet to an ATP requester endpoint. ATP reply packets can have up to eight packets (4624 bytes), and zero-length TSDUs are not supported.

OTRcvUReply

A client of a connectionless transaction-based protocol such as ATP can use the `OTRcvUReply` function to receive an incoming ATP reply packet from an ATP requester endpoint. ATP reply packets can have up to eight packets (4624 bytes), and zero-length TSDUs are not supported.

ATP Reference

This section describes the options that are specific to ATP, defines the constant you use to specify the ATP protocol for option management functions, and indicates the generic Open Transport options you can use with ATP.

Options

There are several ATP-specific options, which are defined as the following:

```
#define ATP_OPT_REPLYCNT    0x2110  /* ATP Reply packet count */
#define ATP_OPT_DATALEN     0x2111  /* ATP packet data Length */
#define ATP_OPT_RELTIMER    0x2112  /* ATP release timer */
#define ATP_OPT_TRANID      0x2113  /* Requests transaction ID */
```

The `ATP_OPT_REPLYCNT` option indicates the number of reply packets in the current ATP reply being received. The `ATP_OPT_DATALEN` option indicates a maximum data packet length that differs from the ATP default of 578; only the PAP server uses this option. The `ATP_OPT_TRANID` option adds the ATP transaction ID added to every request packet.

The `ATP_OPT_RELTIMER` option indicates the amount of time the responder must wait for a transaction release packet before it purges a request entry from its transactions list. Acceptable values are 0 (30 seconds), 1 (1 minute), 2 (2 minutes), 3 (4 minutes), 4 (8 minutes).

In order to use any option with ATP, you must indicate which protocol the option is intended for. To do this, you use a constant for the ATP protocol in the `level` field of the `TOption` structure when you specify an option.

```
#define ATK_ATP         'ATP '
```

ATP also allows you to use the generic Open Transport options `OPT_RETRYCNT` and `OPT_INTERVAL`, which are described in the chapter “Option Management” in this book.

Printer Access Protocol (PAP)

Contents

This chapter describes how Open Transport implements the Printer Access Protocol (PAP). It explains how you can use PAP to set up a printer server endpoint that awaits connection requests from active PAP endpoints. The chapter also explains how to set up an active PAP client endpoint, how to send data directly from it to the printer server, and how the client endpoint receives messages back from the server. PAP offers a connection-oriented transactionless service that is particularly well suited to creating both the client and the server side of a client-server pair of endpoints.

You should read this chapter if you want to write an application that uses PAP to print directly to AppleTalk printers. This chapter explains how you

- create and use an active PAP client endpoint
- create and use a passive PAP server endpoint
- send and receive data with PAP
- divide a PAP data stream into discrete logical units
- set a PAP server to respond to a client's SendStatus call

This chapter begins with a description of PAP and the services that it provides under Open Transport. The section "Using Open Transport Functions With PAP" then gives detailed information about how PAP client applications use the endpoint functions that Open Transport provides for connection-oriented transactionless protocols. For a more detailed explanation of endpoints and their functions, read the chapter "Endpoints" in this book.

For an overview of PAP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter. PAP under Open Transport conforms to the detailed specifications in *Inside AppleTalk*, second edition. See that book for further information about the features mentioned here.

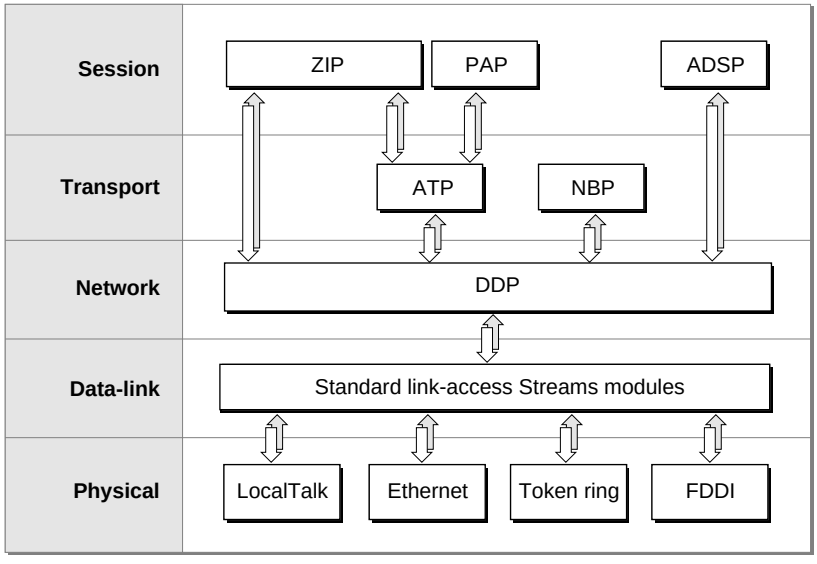
About PAP

The **Printer Access Protocol (PAP)** is an asymmetrical connection-oriented transactionless protocol that enables communication between client and server endpoints, allowing multiple connections at both ends. PAP uses ATP packets to transport the data once a connection is open to the server.

PAP is the protocol that ImageWriter and LaserWriter printers in the AppleTalk environment use for direct printing—that is, when a workstation sends a print job directly to a printer connected to the network instead of using a print spooler. Open Transport PAP provides a single protocol implementation for all AppleTalk printers that is integrated into the AppleTalk protocol stack.

Figure 15-1 shows how a PAP endpoint provider encompasses its underlying delivery protocol and link-access Streams modules.

Figure 15-1 The PAP endpoint provider's underlying delivery mechanism



One of the unique features of PAP is its ability to determine which connection request to honor when there are several requests outstanding at the same time. At any time a PAP server endpoint can receive requests to open a connection from different client endpoints. For example, a printer server is available on a network to many workstations, several of which can send data to the printer at

any time. PAP uses an arbitration scheme to allow a server to accept a connection with the workstation that has been waiting the longest for a connection. The scheme works this way:

1. A PAP server receives a connection request but delays granting it for a predefined length of time (nominally 2 seconds). This default time period is implementation specific and is defined in *Inside AppleTalk,* second edition.
2. The PAP server accumulates any additional connection requests that come in from other endpoints during that time period.
3. At end of the time period, the PAP server obtains the accumulated wait time from each workstation endpoint requesting a connection. The workstations have continued to track the amount of elapsed time spent waiting for access to the server. For example, if a workstation client has to try several times to connect to a busy LaserWriter, the workstation continues to track the total time since the first connection attempt and reports that amount to the LaserWriter on every subsequent connection attempt.
4. The PAP server then grants the request of the workstation that has waited the longest.

Using PAP

To use Open Transport PAP, you first open an endpoint as a PAP endpoint, which causes Open Transport to allocate the memory PAP needs for data buffers and for storing the variables PAP uses to maintain the connection between endpoints. After a connection is established, PAP manages and controls the data flow between the two endpoints throughout a session to ensure that data is delivered and received in the order in which it was sent and that duplicate data is not sent.

Communication between two client applications using PAP occurs over a connection between two endpoints that provides reliable data delivery. When you bind a PAP endpoint, the binding process associates a local protocol address with the endpoint. In PAP, this identifies the socket address, and PAP uses this as part of the address for sending and receiving packets of data. Each socket can maintain concurrent PAP connections with several other sockets, but there can be only one PAP connection between any two sockets at one time.

As with other connection-oriented protocols, Open Transport PAP allows you to create a passive endpoint that listens for incoming connection requests rather than initiating such requests. In addition, the implementation of PAP under Open Transport includes some features that are specific to the two AppleTalk connection-oriented protocols, PAP and ADSP. These are

- an end-of-message option that lets you divide streams of data into logical units
- locally implemented orderly disconnects rather than over-the-wire remote disconnects

A feature unique to PAP is the ability to arbitrate connections requests and to retry attempts to open a connection. This allows PAP printer servers to accept requests from the workstations that have been waiting the longest to print, and it allows PAP workstations to keep trying to get their print request through to the printer.

Binding PAP Endpoints

You have two choices when binding a PAP endpoint: You can create an endpoint that can initiate connections and receive connection requests, or you can create a passive endpoint that can only receive connection requests. Typically, a passive PAP endpoint is a printer server.

If your endpoint can initiate connections, you can bind it as a normal Open Transport endpoint and use any of the three AppleTalk address formats for the socket address: DDP, NBP, or the combined DDP-NBP format. If the bind is successful, the endpoint is ready for use in establishing and using a connection.

The other choice when binding a PAP endpoint is to establish it as a **passive peer** that listens for incoming connection requests. The passive peer can accept or deny a connection request based on criteria that you define. The use of a passive peer is typical of a server environment in which a server, such as a printer server, is registered with a single name. Endpoints throughout the network can contact the printer server with connection requests. The server can accept or deny a request. It might deny a request, for example, when its resources are exhausted.

To create a passive peer that listens, you specify a queue length greater than 0 during the binding process. The number you use determines how many connection requests the endpoint can support. Once the endpoint is bound, it starts listening for incoming connection requests. When a request arrives, the

endpoint retrieves certain information about the request and continues to process the connection request by accepting or rejecting it.

You can bind multiple PAP endpoints to the same socket, but you can have only one passive peer that listens for a given socket. When a server accepts a connection from a client workstation for processing its print request, it cannot accept another connection request from the same workstation endpoint. As with other connection-oriented protocols, you can only have one connection between the same pair of endpoints.

Specifying PAP Options

You can send a PAP data stream that has no logical boundaries within it that need to be preserved across the connection, or you can use **transport service data units (TSDUs)** to separate the data stream into discrete logical units when sending and receiving it across a connection.

By default, PAP does not support TSDUs. Instead, PAP sends and receives a continuous stream of data with no message delimiters, which means you can exchange data with an endpoint whose protocol does not support TSDUs. If you do not specify any PAP-specific options, your packets are not restricted to Open Transport PAP endpoints, and you can provide transport-independent data transmission.

The Enable End-of-Message Option

If transport independence is not crucial for your application, you can use a PAP-specific option that allows TSDUs. The `OPT_ENABLEEOM` option enables the PAP end-of-message feature, which permits dividing data streams into smaller logical units. Open Transport uses a flag in the send and receive functions to indicate multiple sends and receives. The use of this flag, the `T_MORE` flag, allows you to break up a large data stream without losing its logical boundaries at the other end of the connection. The flag, however, indicates nothing about how the data is packaged for transport on the lower-level protocols below the PAP endpoint provider.

To send a data stream that is broken up into TSDUs, you set the `T_MORE` flag on each send. This indicates to the remote connection end that there are more packets coming that are part of this same data stream. When a packet arrives without the `T_MORE` flag set, the remote end knows this is the last packet for this stream of data. It is possible for this last packet to contain no data because PAP supports the sending of zero-length packets. This could occur when you send a

packet with the T_MORE flag set only to discover that you have no more data to send. In this case, PAP still expects another packet, but you have no data to put into it. You can send a zero -length packet to set the T_MORE flag correctly.

Because printers expect an EOM indicator on the last packet, if you do not choose to use the OPT_ENABLEEOM option, PAP takes care of that for you, guaranteeing that the EOM indicator is set on the last packet. If, however, you do choose to use the OPT_ENABLEEOM option, you are responsible for setting the EOM indicator, by using the T_MORE flag on every packet but the last.

When you use TSDUs with PAP, you cannot change the size of the TSDU after you have established the connection with another endpoint. This means that you don't need to double-check the TSDU size after the first packet because it will always be the same for all packets using this connection.

The Open Retry Option

When a PAP endpoint provider calls the OTConnect function, which creates a connection request, by default it does not try a second time to establish the connection. The PAP default for this option is a value of 1, which refers to the number of times that PAP tries to open a connection, which means that PAP permits only the initial OTConnect request. (Given the effect of its default value, you might find it easier to think of this option as the *try* open connection option, rather than as the *retry* open connection option.)

Note
The default value of 1 for this option differs from the default retry count of 5 specified in *Inside AppleTalk*, second edition. In other aspects of Open Transport AppleTalk, AppleTalk protocols adhere to the specifications detailed in that book. ◆

To force PAP to try again to open the connection, you can use a value greater than 1 for the PAP_OPT_OPENRETRY option. Note that if you change the value of this option, you compromise the transport independence of your application. Thus, if you need to retry sending the open connection packets, you must deliberately choose to use the option and make your code transport dependent. Workstation client applications that want to print data, for example, will probably keep trying to get access to a printer server, retrying printer connections until the user presses the cancel button. Once the user requests a halt, of course, PAP stops trying to connect.

The Server Status Option

In a client-server interaction, a client may sometimes need to know the status of the server. In these cases, the client can request the server's status. This request can occur outside a connection. If the `OPT_SERVERSTATUS` option has been set, with a C string 255 bytes long, the server can return that string as the server status.

Disconnecting

As with all connection-oriented Open Transport protocols, PAP supports abortive disconnects. In addition, PAP supports orderly disconnects, although it can only implement them locally.

An abortive disconnect directs the remote endpoint to abruptly tear down its connection without making any accomodation for the data that may be in the transmission pipeline at the time. You can define your own handshake to prevent losing data during the disconnect process.

PAP implements orderly disconnects locally, not over the wire. This means that immediately after you request the disconnect, PAP sends all data buffered at the local end and then tears down the connection, breaking communication with the remote end. As a result, no data can be sent from either the local or remote endpoint. The endpoints can continue to process data already in their receive queues, but no new data can go out.

Using General Open Transport Functions With PAP

This section describes any special considerations you must take into account for Open Transport functions when you use them with the Open Transport PAP implementation. These may mean using specific parameter values or using specific Open Transport functions. For example, PAP only works with two of the Open Transport sending and receiving functions, `OTSnd` and `OTRcv`, because only these work with protocols that are connection-oriented and transactionless. You must be familiar with the function descriptions in the chapter "Endpoints" in this book before reading this section.

OTBind

The `OTBind` function associates a local protocol address with the endpoint specified by the `ref` parameter.

You can bind multiple PAP endpoints to a single protocol address, but you can bind only one passive endpoint that listens at that address.

With PAP, as with other connection-oriented protocols, the `req->qlen` parameter specifies the number of outstanding connection requests that an endpoint can support. The endpoint can negotiate the final value of `qlen` if it cannot handle the requested number of outstanding connection requests, but in PAP, the value of `qlen` cannot be negotiated to 0 from a requested value greater than 0.

OTConnect

The `OTConnect` function request a connection to a specified remote endpoint.

PAP does not allow application-specific data to be included when you establish a connection, so you need to set the `sndcall->udata.len` field to 0. PAP ignores the `sndcall->udata.buf` field.

OTRcvConnect

The `OTRcvConnect` function reads the status of a previously issued connection request.

Because PAP does not allow application-specific data to be associated with a connection request, you need to set the `call->udata.len` field to 0. PAP ignores the `call->udata.buf` field.

OTListen

The `OTListen` function listens for an incoming connection request.

PAP does not allow application-specific data to be included when you request a connection, so you need to set the `call->udata.len` field to 0. PAP ignores any data in the `call->udata.buf` field.

OTAccept

The `OTAccept` function accept a connection request either on the same endpoint that received the connection request or on a different endpoint.

PAP does not allow application-specific data to be included when you accept a connection, so you need to set the `call->udata.len` field to 0. PAP ignores the `call->udata.buf` field.

OTSnd

The `OTSnd` function sends normal and expedited data through a connection-oriented transactionless endpoint.

PAP supports TSDUs through the `OPT_ENABLEEOM` option, although its use compromises the transport independence of your application. In PAP, TSDUs sent from the client endpoint can be of infinite length, but TSDUs sent from a server endpoint can only be up to 512 bytes long. Zero-length packets are supported by PAP.

OTRcv

The `OTRcv` function receives normal and expedited data through a connection-oriented transactionless endpoint.

PAP supports TSDUs through the `OPT_ENABLEEOM` option, although its use compromises the transport independence of your application. In PAP, TSDUs sent from the client endpoint can be of infinite length, but TSDUs sent from a server endpoint can only be up to 512 bytes long. Zero-length packets are supported by PAP.

OTSndDisconnect

The `OTSndDisconnect` function initiates an abortive disconnect or rejects a connection request.

In an abortive disconnect, the `call` parameter is ignored because PAP does not allow application-specific data to be associated with a disconnect. You need to set the `call->udata.len` field to 0. PAP ignores the `call->udata.buf` field.

OTRcvDisconnect

The `OTRcvDisconnect` function returns information about why a connection attempt failed or an established connection was terminated.

Because PAP does not allow application-specific data to be associated with a disconnect, you need to set the `discon->udata.len` field to 0. PAP ignores in the `discon->udata.buf` field.

PAP Reference

This section describes the `PAP_OPT_OPENRETRY` option that is specific to PAP, defines the constant you use to specify the PAP protocol for option management functions, and indicates the generic Open Transport options that you can use with PAP.

Options

This section describes the PAP-specific option that you can use with the `OTConnect` provider function.

The only option that is specific to PAP is the open retry option, which is defined as following:

```
#define PAP_OPT_OPENRETRY       0x2120      /* PAP open retry count */
```

This option forces PAP to try again to open a connection that PAP has tried once already to establish without success.

In order to use any option with PAP, you must indicate which protocol the option is intended for. To do this, you use a constant for the PAP protocol in the `level` field of the `TOption` structure when you specify an option.

```
#define ATK_PAP         'PAP '
```

With PAP, you can use the generic `OPT_ENABLEEOM`, `OPT_CHECKSUM` and `OPT_SERVERSTATUS` options, which are further described in the chapter “Option Management” in this book.

Serial Endpoint Providers

Contents

This chapter describes how you can use serial endpoint providers to transfer data between devices connected to a modem or printer port. Open Transport supports asynchronous serial data communication between client applications through these ports. This chapter provides information about Open Transport functions and options that are specific to serial endpoint providers. You need this information only if you have a specific need to use serial communication.

Serial endpoints provide low-level support for communicating with serial devices that cannot be accessed through the Communications Toolbox or Printing Manager; for example, a scientific instrument or a printer that does not support QuickDraw. Before you decide to use a serial endpoint, you should determine whether it is the appropriate solution for your communications needs.

To get the most out of this chapter, you should already be familiar with the concepts and application interfaces described in the chapters "Introduction to Open Transport," "Providers," "Endpoints," "Option Management," and "Configuration Management" in this book. For information about the Macintosh serial port hardware, including circuit diagrams and signal descriptions, see *Guide to the Macintosh Family Hardware*, second edition. The Open Transport serial interface software modules are based on the UNIX STREAMS standard. For more information about STREAMS, see *UNIX System V Release 4: Programmer's Guide: STREAMS*. The Open Transport API is based on the XTI standard as documented in *X/Open CAE Specification: X/Open Transport Interface (XTI)*.

This chapter begins with a brief summary of key concepts in serial data communication, then describes how you can use serial endpoint providers to

- configure a serial port
- send and receive data through a serial port
- interpret serial communication status information

The section "Using General Open Transport Functions With Serial Endpoints," beginning on page 16-14 describes serial-specific information relating to functions described in the "Endpoints" chapter of this book and the section "Using Options to Change Serial Communications Settings," beginning on page 16-11 describes the options you can specify when you configure a serial endpoint provider. The reference section describes those constants, options, and `OTIoctl` function commands available to users of Open Transport serial endpoint providers.

About Serial Endpoint Providers

Open Transport serial endpoint providers provide full-duplex low-level support for asynchronous, interrupt-driven serial data transfers through the modem and printer ports. Serial endpoint providers use connection-oriented data streams. They do not support the functions that provide connectionless or transaction-based service. Because of the point-to-point nature of serial communications, there are a few differences between using a serial endpoint and using other connection-oriented endpoints.

One of the key differences is that there are no addresses for serial endpoints because serial communications is point-to-point. As such, no addressing information is possible and all address parameters for serial endpoint functions need to be set to zero.

The other important difference is that only one serial endpoint can own the hardware at a given time. That is, only one serial endpoint provider can initiate and accept a connection on a given port at a time, although there can be several listening endpoints on a given port simultaneously.

About Serial Communication

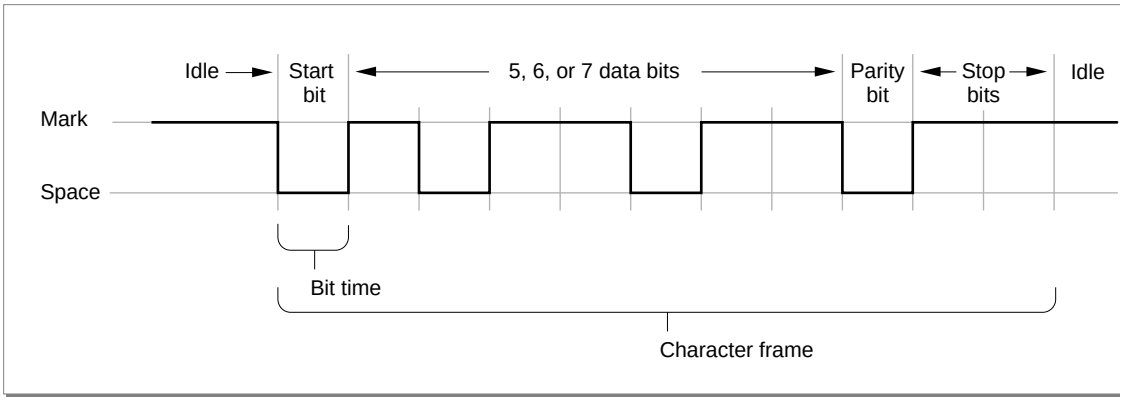
Open Transport serial communication, like any data transfer between endpoints, requires coordination between the sender and receiver; for example, when to start the transmission and when to end it, when one particular bit or byte ends and another begins, when the receiver's capacity has been exceeded, and so on. The scope of serial data transmission protocols is large and complex, encompassing everything from electrical connections to data encoding methods.

This section provides a brief overview of the protocol that governs the lowest level of data transmission—how serialized bits are sent over a single electrical line.

When a sender is connected to a receiver over an electrical connecting line, the line is initially in an idle state, called the **mark state,** which has a positive voltage level. Changing the state of the line by shifting the voltage to a negative value creates what is called a **space.** Once this change has occurred, the receiver interprets a negative voltage level as a 0 bit, and a positive voltage level as a 1 bit. These transitions are shown in Figure 16-1.

The change from the mark state to a space is known as the **start bit,** and this triggers the synchronization necessary for asynchronous serial transmission. The start bit delineates the beginning of the transmission unit defined as a **character frame.** The receiver then samples the voltage level at periodic intervals, known as the **bit time,** to determine whether a 0 bit or a 1 bit is present on the line.

Figure 16-1 The format of serialized bits



The bit time is expressed in samples per second, known as the **baud rate.** The baud rate must be agreed upon by sender and receiver before transmitting data in order for a successful transfer to occur. Common values are 1200 baud and 2400 baud. In the case where one sampling interval can signal a single bit, a baud rate of 1200 results in a transfer rate of 1200 bits per second (bps). Note that because modern protocols can express more than one bit value within the sampling interval, the baud rate and the transfer rate may not be identical.

Before transmission, the sender and receiver also agree on a serial data format; that is, how many bits of data constitute a character frame and what happens after those bits are sent. Open Transport serial endpoints support frames of 5, 6, 7, or 8 bits in length. Character frames of 7 or 8 data bits are commonly used for transmitting ASCII characters.

After the data bits in the frame are sent, the sender can optionally transmit a parity bit for error-checking. There are various parity schemes, which the sender and receiver must agree upon prior to transmission. In odd parity, a bit

is sent so that the entire frame always contains an odd number of 1 bits. Conversely, in even parity, the parity bit results in an even number of 1 bits. No parity means that no additional bit is sent.

To signify the end of the character frame, the sender places the line back to the mark state for a minimum specified time interval. This interval has one of several possible values: 1 bit time, 2 bit times, or 1-1/2 bit times. This signal is known as the **stop bit,** and returns the transmission line back to the mark state.

Electrical lines are always subject to environmental perturbations known as **noise.** This noise can cause errors in transmission by altering voltage levels so that a bit is reversed, shortened, or lengthened. When this occurs, the ability of the receiver to distinguish a character frame may be affected, resulting in a framing error.

The **break signal** is a special signal that falls outside the character frame. The break signal occurs when the line is switched from the mark state to a space and held there for longer than a character frame. The break signal resembles an ASCII NUL character (a string of 0-bits), but exists at a lower level than the ASCII encoding scheme that governs the encoding of information within the character frame.

DTR and CTS Signals

The electrical characteristics of a serial communications connection are specified by various interfacing standards. The specifications of these standards are contained in documents available from the Electronic Industries Associations (EIA) that cover aspects of the connection, such as its electrical signal characteristics and its interface circuits.

The principal signals used by Open Transport serial endpoint providers are the **Data Terminal Ready (DTR)** and **Clear To Send (CTS) signals.** In Macintosh computers, these two signals are connected to each other. These signals are described in the following bullets. Note that in these definitions, the term *data terminal equipment (DTE)* is used to describe the initiator or controller of the serial connection, typically the computer. The term *data communication equipment (DCE)* describes the device that is connected to the DTE, such as a

modem or printer. For specific information about how these signals are used in Macintosh computers, see *Guide to the Macintosh Family Hardware*, second edition.

- The Data Terminal Ready (DTR) signal indicates that the DTE (that is, your computer) is ready to communicate. Deasserting this signal causes the DCE (that is, your modem or printer) to suspend transmission. The DTR signal is the most important control line for a modem because when it is deasserted, most modem functions cease and the modem disconnects from the telephone line.
- The Clear To Send (CTS) signal indicates that the DCE (your modem or printer) is ready to send data. Since most communications between microcomputers are full-duplex nowadays, the CTS signal is permanently asserted.

Asynchronous and Synchronous Communication

Serial data transfers depend on accurate timing in order to differentiate bits in the data stream. This timing can be handled in one of two ways: asynchronously or synchronously. In asynchronous communication, the scope of the timing is a single byte. In synchronous communication, the timing scope comprises one or more blocks of bytes. The terms *asynchronous* and *synchronous* are slightly misleading because both kinds of communication require synchronization between the sender and receiver.

Asynchronous communication is the prevailing standard in the personal computer industry because it is easier to implement and because it has the unique advantage that bytes can be sent whenever they are ready, as opposed to waiting for blocks of data to accumulate.

IMPORTANT

Do not confuse asynchronous communication with asynchronous execution. *Asynchronous communication* is a protocol for coordinating serial data transfers. *Asynchronous execution* refers to the capability of a device driver to carry out background processing. Serial endpoints support both asynchronous communication and asynchronous execution. ▲

Open Transport serial endpoints do not support synchronous communications protocols. However, they do support synchronous clocking supplied by an external device.

Handshaking Methods for Flow Control

Because a sender and receiver can't always process data at the same rate, some method of negotiating when to start and stop transmission is required. Open Transport serial endpoint providers support two methods of controlling serial data flow, known as **handshaking.** One method relies on the serial port hardware, the other is implemented in software.

Hardware handshaking uses two of the serial port signal lines to control data transmission. When the serial endpoint provider is ready to accept data from an external device, it asserts the Data Terminal Ready (DTR) signal on pin 1 of the serial port, which the external device receives through its Clear To Send (CTS) input. Likewise, the Macintosh receives the external device's DTR signal through the CTS input on pin 2 of the serial port. When either the Macintosh or the external device is unable to receive data, it negates its DTR signal, and the sender suspends transmission until the signal is asserted again.

Software handshaking uses an agreed-upon set of characters for the start and stop signals. Open Transport serial endpoints support XON/XOFF handshaking, which typically assigns the ASCII DC1 character (Control-Q) as the start signal and the DC3 character (Control-S) as the stop signal, although you can choose different characters.

Using Serial Endpoints

Serial endpoint providers use standard Open Transport functions for binding, requesting and accepting connections, sending and receiving data, and managing options. You can send and receive the desired data using the standard Open Transport `OTSnd` and `OTRcv` functions. You can call these functions either synchronously or asynchronously, as described in the chapter "Endpoints" in this book.

In addition, Open Transport provides specialized serial-specific commands and options that allow you to

- set the flow-control handshaking
- use an external timing signal for synchronous clocking
- set or clear a break signal
- get status information about a port and any associated transmission errors
- define how characters with parity errors are handled
- request burst mode operation
- define receive timeout options
- set the framing type

Opening and Closing Serial Endpoints

To open serial endpoints, you need to supply a configuration string to the `OTOpenEndpoint` function by using one of the following constants:

Constant name	String value	Description
`kSerialName`	`"serial"`	Default serial port
`kSerialPortAName`	`"serialA"`	Serial port A
`kSerialPortBName`	`"serialB"`	Serial port B
`kSerialPortABName`	`"serialAB"`	Serial port AB

For example, the following line of code opens a serial endpoint on serial port A:

```
OTOpenEndpoint(OTCreateConfiguration(kSerialPortAName));
```

To close a serial endpoint provider, you use the standard Open Transport function `OTCloseProvider`, described in the chapter “Providers” in this book.

Sending and Receiving Data

As with all endpoints, you must call the `OTBind` function before you can use a serial endpoint provider to send or receive data. For serial endpoint providers that initiate outgoing data, you need to bind with a queue length (the `qlen`

parameter) of 0. When you wish to start transferring data, you must call the `OTConnect` function to place the endpoint in the data transfer state and allow you to call the `OTSnd` and `OTRcv` functions. Calling the `OTSndDisconnect` function releases the connection.

For serial endpoint providers that listen for incoming data, you need to bind with a queue length of 1. You cannot bind with a queue length greater than 1. When an incoming character is detected on the serial port, you receive a connect indication. You can accept the indication on the current endpoint, or you can accept it on another serial endpoint, which has a queue length of 0 or which is not yet bound. In either case, once the accepting endpoint returns to the `T_IDLE` state, the original endpoint once again listens for incoming data and gets a connect indication if another incoming character is detected. Calling the `OTSndDisconnect` function releases the connection and allows your endpoint to continue listening on the port. Your endpoint can continue to listen until you call the `OTUnbind` function.

You can create a number of serial endpoints to listen on a given serial port, but only one can have a connection at a time. The first serial endpoint to connect owns the hardware; other endpoints that subsequently attempt to connect receive a `kOTAddressBusyErr` result code.

Using Serial-Specific Commands

You can control several aspects of serial communication by using the Open Transport function `OTIoctl` with different serial-specific commands. The `OTIoctl` function, described in the chapter "Providers" in this book, accesses the low-level serial driver control and status functions (`PBControlAsync` or `PBStatusAsync`). For information about Device Manager functions for opening, closing, and communicating with device drivers, see the book *Inside Macintosh: Devices*.

You can assert the DTR signal for the serial port by using a value of `kOTSerialSetDTROn` with the `I_SetSerialDTR` command and you can negate it with a value of `kOTSerialSetDTROff`. Likewise, you can use the `I_SetSerialBreak` command to set or negate the break signal with values of `kOTSerialSetBreakOn` and `kOTSerialSetBreakOff` or you can use a number greater than 1 to indicate the number in milliseconds to assert a break signal temporarily.

You can also use the `OTIoctl` function commands to set the XOFF state of the serial port and to indicate whether the port is to send an XOFF or XON

character. Using a value of `kOTSerialForceXOffTrue` with the `I_SetSerialXOffState` command sets the XOFF state of the serial port, which is equivalent to receiving an XOFF character, and using a value of `kOTSerialForceXOffFalse` with this command clears the XOFF state, which is equivalent to receiving an XON character.

Using a value of 1 with the `I_SetSerialXOn` and `I_SetSerialXOff` commands causes the serial port to unconditionally send an XON or XOFF character, respectively. A value of 0 with these functions causes the character to be sent only if the last input flow-control character sent was the opposite kind—that is, the XOFF or XON character, respectively.

Using Options to Change Serial Communications Settings

Serial endpoints currently support six options. These options are defined by the XTI-level constant `COM_SERIAL`, which has a value of `'SERL'`.

When you open a serial endpoint, Open Transport configures the selected port with the default settings of 19200 baud, 8 data bits per character, no parity bit, 1 stop bit, and no handshaking. You can change these settings using various options, all of which use 4-byte unsigned integer values. There is also a serial status option that provides current information about the serial port. Four of the options are fairly straightforward and are described here; using the other two options is more complicated, and their use is described in the two subsequent sections.

- The baud rate option sets the serial baud rate.The serial module chooses the closest baud rate supported that matches the requested rate. Possible values range from 300 to 57600 baud transmission rates (depending on the hardware capability). The default value is 19200 baud.
- The data bits option selects the number of data bits to be used. Legal values are 5, 6, 7, and 8. The default value is 8 data bits.
- The stop bits option selects the number of stop bits to be used. This value corresponds to ten (10) times the actual number of stop bits. Legal values are 10, 15, and 20, which correspond to stop bits of 1, 1.5, and 2. The default value is 10, which is equivalent to 1 stop bit.
- The parity option selects the parity to be used. Legal values are `kOTNoParity 0)`, `kOTOddParity(1)`, and `kOTEvenParity(2)`. The default value is `kOTNoParity`.
- The receive timeout option sets the number of milliseconds the receiver should wait to receive more data before timing out.

- The error character option defines how characters with parity errors are handled—that is, if they are replaced and with which character. Open Transport provides the define statements (and C++ inline functions), `OTSerialSetErrorCharacter` and `OTSerialSetErrorCharacterWithAlternate`, to help place the character bits correctly.
- The external clock option requests an external clock. This option may not be supported by all serial drivers.
- The burst mode option requests that the serial driver continues looping, reading incoming characters, rather than waiting for an interrupt for each character. This option may not be supported by all serial drivers.

Setting Flow-Control Handshaking

This option selects the flow-control handshaking to be used by the serial endpoint providers. The handshaking can be either hardware, using the DTR and CTS signals, or software, using the XON and XOFF characters. The default value of this option is no handshaking.

A schematic diagram of this 4-byte option value looks like this:

```
xxxxxxxxxxxxxxxx        xxxxxxxx            xxxxxxxx
handshake bitmap        XON character       XOFF character
```

The high word is a bitmap with one or more bits set, indicating the type of handshaking requested. The DTR signal is normally asserted when the serial endpoint is opened and negated when it is closed. The CTS signal is normally always asserted. If the XON and XOFF character values are 0 and if XON/OFF handshaking was requested, Open Transport uses the default values of Control-S for XOFF and Control-Q for XON.

Open Transport provides a define statement and a C++ inline function (`SerialHandshakeData`) that you can use to create the 4-byte option value.

Obtaining Status Information About the Serial Port

The serial status option is a read-only option that returns status information on the serial port. It is a 4-byte unsigned integer containing a bitmap that can provide the following information about errors or changes in status that may have occured:

- A hardware overrun has occurred due to an overflow of the hardware input buffer.
- A software overrun has occurred due to an overflow of the software input buffer.
- A parity error has occurred due to the serial hardware detecting an incorrect parity bit.
- A framing error has occurred due to the serial hardware detecting a stop bit error.
- A break has occurred on the line, and the break signal has been asserted.
- The endpoint provider has sent an XOFF character, which initiates flow control.
- The endpoint provider has negated the DTR signal, which initiates flow control.
- The endpoint provider has negated the CTS signal, which initiates flow control.
- The endpoint provider has received an XOFF character, and so all output is on hold.
- The endpoint provider has initiated a break that is still in progress.

Data received from the serial port passes through a hardware buffer and then into a software buffer managed by the input driver for the port. Each input driver's buffer can initially hold up to 64 characters, but you can specify a larger buffer with standard Open Transport functions. This is normally not necessary because Open Transport provides additional buffering as part of its processing.

Because the serial hardware in some Macintosh computers relies on processor interrupts during I/O operations, overrun errors are possible if interrupts are disabled while data is being received at the serial port. To prevent such errors, the Disk Driver and other system software components are designed to store any data received by the modem port while they have interrupts disabled and then pass this data to the port's input driver. Because the system software only monitors the modem port, the printer port is not recommended for two-way communication at data rates above 300 baud.

Overrun, parity, and framing errors are usually handled by requesting that the sender retransmit the affected data. Break errors are typically initiated by the client application, which handles them as appropriate.

Using General Open Transport Functions With Serial Endpoints

This section describes any special considerations that you must take into account for Open Transport functions when you use them with serial endpoint providers. You should be familiar with the function descriptions in the chapter "Endpoints" in this book before reading this section.

Obtaining Endpoint Data With Serial Endpoints

This section describes the possible values you can get for endpoint information when using a serial endpoint.

OTOpenEndpoint, OTAsyncOpenEndpoint, and OTGetEndpointInfo

The following values can be returned by the `info` parameter to the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, and `OTGetEndpointInfo` functions when used with serial endpoint providers:

Parameter	Serial	Meaning
`info->addr`	0	Addresses are not used.
`info->options`	Greater than 0	Maximum number of bytes needed to hold protocol-specific options.
`info->tsdu`	`T_INVALID`	TSDUs are not supported.
`info->etsdu`	`T_INVALID`	Transfer of expedited data is not supported.
`info->connect`	`T_INVALID`	Data cannot be sent with functions that establish connections.
`info->discon`	`T_INVALID`	Data cannot be sent with abortive disconnects.
`info->servtype`	`T_COTS`	Orderly disconnects are not supported.
`info->flags`	-	No flags are set.

IMPORTANT

The values shown in the preceding table are subject to change. Be sure to to use the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, or `OTGetEndpointInfo` function to obtain the current values for these parameters. ▲

These fields and the significance of their values are described in more detail in the document *X/Open CAE Specification: X/Open Transport Interface (XTI)*.

Using Endpoint Functions With Serial Endpoints

This section describes serial-specific information about functions described in the chapter "Endpoints" in this book.

OTBind

The `OTBind` function associates a serial port with the endpoint you specify. Because serial communication is point-to-point over a hardware connection, you cannot specify an address. Therefore, you must specify 0 as the length of the address in the `reqaddr->TBind.addr.len` parameter. You can bind multiple serial endpoints to listen at a single port.

With serial endpoints, the `req->qlen` parameter, which specifies the number of outstanding connection requests that an endpoint can support, can only have a value of 0 or 1. To listen, a serial endpoint provider must have a queue length value of 1; to accept connections, the endpoint can have a value of 0 or 1. A value greater than 1 results in a error code.

OTConnect

The `OTConnect` function requests a connection to a specified remote endpoint.

Because serial endpoint providers do not allow you to send any application-specific data during the connection establishment phase, you must set the `sndcall->udata.len` field to 0. Serial endpoints ignore any data in the `sndcall->udata.buf` field.

OTListen

The `OTListen` function listens for an incoming connection request.

Serial endpoints do not allow application-specific data to be included when you request a connection, so you need to set the `call->udata.len` field to 0. Serial endpoints ignore the `call->udata.buf` field.

OTAccept

The `OTAccept` function accepts a connection request. You can accept a connection either on the same endpoint that received the connection request or on a different endpoint.

Serial endpoints do not allow application-specific data to be included when you accept a connection, so you need to set the `call->udata.len` field to 0. Serial endpoints ignore the `call->udata.buf` field.

OTSnd

The `OTSnd` function sends data through a connection-oriented transactionless endpoint. Serial endpoints do not support TSDUs.

OTRcv

The `OTRcv` function receives data through a connection-oriented transactionless endpoint. Serial endpoints do not support TSDUs.

OTSndDisconnect

The `OTSndDisconnect` function initiates an abortive disconnect or rejects a connection request.

In an abortive disconnect, the `call` parameter is ignored because serial endpoints do not allow application-specific data to be associated with a disconnect. You need to set the `call->udata.len` field to 0. Serial endpoints ignore the `call->udata.buf` field.

OTRcvDisconnect

The `OTRcvDisconnect` function returns information about why a connection attempt failed or an established connection was terminated.

Because serial endpoints do not allow application-specific data to be associated with a disconnect, you need to set the `discon->udata.len` field to 0. Serial endpoints ignore the `discon->udata.buf` field.

Serial Endpoint Providers Reference

This section describes the constants, options, and serial-specific commands used by Open Transport serial endpoint providers.

Constants

This section describes the constants used by serial endpoints. You can use the constant names `kSerialName`, `kSerialPortAName`, `kSerialPortBName`, and `kSerialABName` when calling the `OTCreateConfiguration` function to configure a serial endpoint. This function is described in the chapter "Configuration Management" in this book.

```
#define kSerialName                 'serial'        /* Default serial port */
#define kSerialPortAName            'serialA'       /* Serial port A */
#define kSerialPortBName            'serialB'       /* Serial port B*/
#define kSerialPortABName           'serialAB'      /* Serial port AB*/
```

You use the values in the next enumeration to define the type of framing your serial port is using. These values are used in the `fCapabilities` field in the `OTPortRecord` structure, described in the chapter "Configuration Management" in this book.

```
enum{
    kOTSerialFramingAsync        = 0x01, /* Supports asynchronous serial framing */
    kOTSerialFramingHDLC         = 0x02, /* Supports serial HDLC framing */
    kOTSerialFramingSDLC         = 0x04, /* Supports serial SDLC framing */
    kOTSerialFramingAsyncPackets = 0x08, /* Supports async packet serial mode */}
```

The `OTIoctl` commands use many constants:

```
kOTSerialSetDTROn               = 1             /* Turn the DTR signal on */
kOTSerialSetDTROff              = 0             /* Turn the DTR signal off */

kOTSerialSetBreakOn             = 0xffffffff    /* Turn the break signal on */
kOTSerialSetBreakOff            = 0             /* Turn the break signal off */
```

```
kOTSerialForceXOffTrue          = 1             /* Unconditional set XOFF state */
kOTSerialForceXOffFalse         = 0             /* Unconditional clear XOFF state */

kOTSerialSendXOnAlways          = 1             /* Always send XON character */
kOTSerialSendXOnIfXOffTrue      = 0             /* Send XON char only if XOFF state */

kOTSerialSendXOffAlways         = 1             /* Always send XOFF character */
kOTSerialSendXOffIfXOnTrue      = 0             /* Send XOFF char only if XON state */
```

This define statement, which is identical to the C++ inline function Open Transport provides for this task, creates the 4-byte option value you use for the `SERIAL_OPT_HANDSHAKE` option:

```
#define SerialHandshakeData(type, onChar, offChar)\
    ((((UInt32)type) << 16) | (((UInt32)onChar) << 8) | offChar)
```

These define statements, which are similar to the C++ inline functions Open Transport provides for these tasks, set the correct placement for the characters you use with the `SERIAL_OPT_ERRORCHARACTER` option:

```
#define OTSerialSetErrorCharacter(rep) \
    ((rep) & 0xff)

#define OTSerialSetErrorCharacterWithAlternate(rep, alternate) \
    ((((rep) & 0xff) | (((alternate) & 0xff) << 8)) | 0x80000000L)
```

This enumeration lists the default values for serial endpoint providers:

```
enum{
    kOTSerialDefaultBaudRate    = 19200,            /* 19200 baud rate */
    kOTSerialDefaultDataBits    = 8,                /* 8 data bits */
    kOTSerialDefaultStopBits    = 10,               /* 1 stop bit */
    kOTSerialDefaultParity      = kOTSerialNoParity, /* no parity */
    kOTSerialDefaultHandshake   = 0,                /* no handshaking */
    kOTSerialDefaultOnChar      = ('Q' & ~0x40),    /* XON = Control-Q */
    kOTSerialDefaultOffChar     = ('S' & ~0x40),    /* XOFF = Control-S */
    kOTSerialDefaultSndBufSize  = 128,              /* send buffer = 128 characters */
    kOTSerialDefaultRcvBufSize  = 128,              /* recv buffer = 128 characters */
    kOTSerialDefaultSndLoWat    = 96,               /* send low-water mark */
```

```
    kOTSerialDefaultRcvLoWat    = 1                 /* recv low-water mark */
    kOTSerialDefaultRcvTimeout  = 10                /* recv timeout, in seconds*/
};
```

Options

This section describes the serial-specific options that you can use with provider functions such as `OTOptionManagement` and `OTConnect`.

Protocol Level

You use this XTI constant when calling the `OTOptionManagement` function to establish the protocol type for an option you are using. You specify this value in the `level` field of the `TOption` stucture. This function and structure are described in the chapter "Option Management" in this book.

```
enum {
    COM_SERIAL          = = 'SERL'
};
```

Serial Options

You can use these options with a protocol level of `COM_SERIAL`. The `SERIAL_OPT_STATUS` option is read only; none of these are association-related options.

```
enum {
    SERIAL_OPT_BAUDRATE                 0x0100,
    SERIAL_OPT_DATABITS                 0x0101,
    SERIAL_OPT_STOPBITS                 0x0102,
    SERIAL_OPT_PARITY                   0x0103,
    SERIAL_OPT_STATUS                   0x0104,
    SERIAL_OPT_HANDSHAKE                0x0105,
    SERIAL_OPT_RCVTIMEOUT               0x0106,
    SERIAL_OPT_ERRORCHARACTER           0x0107,
    SERIAL_OPT_EXTCLOCK                 0x0108,
    SERIAL_OPT_BURSTMODE                0x0109
};
```

Option descriptions

`SERIAL_OPT_BAUDRATE`
Sets the baud rate. Values can be 300 to 56700. (The default is 19200.)

`SERIAL_OPT_DATABITS`
Sets the data bits. Values can be 5, 6, 7, and 8. (The default is 8.)

`SERIAL_OPT_STOPBITS`
Sets the stop bits. Values can be 10, 15, or 20. These reflect a number that is 10 times the bit time value: 1, 1.5, and 2. (The default is 10.)

`SERIAL_OPT_PARITY` Sets the parity. (The default is no parity.)

Parity	Value	Description
`kOTSerialNoParity`	0	No parity.
`kOTSerialOddParity`	1	Odd parity.
`kOTSerialEvenParity`	2	Even parity.

`SERIAL_OPT_STATUS` Returns the current status in this read-only option. One or more bits can be set.

Status	Value	Description
`kOTSerialSwOverRunErr`	`0x01`	Software overrun.
`kOTSerialBreakOn`	`0x08`	A break on the line.
`kOTSerialParityErr`	`0x10`	Parity error.
`kOTSerialOverrunErr`	`0x20`	Hardware overrun.
`kOTSerialFramingErr`	`0x40`	Framing error.
`kOTSerialXOffSent`	`0x0010000`	The XOFF character has been sent.
`kOTSerialDTRNegated`	`0x0020000`	The DTR signal is negated.
`kOTSerialCTLHold`	`0x0040000`	The CTS signal is negated, causing a hold.
`kOTSerialXOffHold`	`0x0080000`	The XOFF character has been received, causing a hold.
`kOTSerialOutputBreakOn`	`0x1000000`	A break has been initiated.

SERIAL_OPT_HANDSHAKE

Sets the handshake to be used by the serial line. (The default is no handshake.) The high word of the integer is a bitmap with 1 or more of the following bits set:

Handshake	Value	Description
kOTSerialXOnOffInputHandshake	1	XON/XOFF set for input.
kOTSerialXOnOffOutputHandshake	2	XON/XOFF set for output.
kOTSerialCTSInputHandshake	4	CTS set on input.
kOTSerialDTROutputHandshake	8	DTR set on output.

The third byte in the option is the XON character. and the lowest byte is the XOFF character. If these two values are 0 and if XON/XOFF handshaking was requested, Open Transport uses the default values of Control-Q for XON and Control-S for XOFF.

SERIAL_OPT_RCVTIMEOUT

Sets the number of milliseconds the receiver should wait before delivering less than the RcvLoWat number of incoming serial characters. If RcvLoWat is 0, then the value is the number of milliseconds of quiet time (no characters being received) that must elapse before characters are delivered to the client. In all cases, this option is advisory and serial drivers are free to deliver data whenever they deem it convenient. For instance, many serial drivers deliver data whenever 64 bytes have been received because 64 bytes is the smallest STREAMS buffer size. Be sure to look at the return value of the option to determine

what it was negotiated to. Here are some examples of its use:

RcvTimeout	RcvLoWat	Action
0	0	Data is delivered immediately after it arrives
x	0	Data is delivered after *x* milliseconds of no incoming characters on the line.
x	y	Data is delivered after *y* characters are received, or *x* milliseconds after the first character is received, whichever comes first.

`SERIAL_OPT_ERRORCHARACTER`

Sets how characters with parity errors are handled. A 0 value disables their replacement. A single character value in the low byte designates the replacement character. When characters are received with a parity error, they are replaced by this specified character. If a valid incoming character matches the replacement character, then the received character's most-significant-bit is cleared. For this situation, an alternate replacement character may be specified in bits 8 through 15 of the 32-bit value, with `0xff` being placed in bits 16 through 23. You can use the macros `OTSrlSetPEChar` and `OTSrlSetPECharWithAlternate` to get the bit placement correct. In this case, whenever a valid character is received that matches the first replacement character, it is replaced with this alternate character (which may be 0).

`SERIAL_OPT_EXTCLOCK`

Requests an external clock. A 0-value turns off external clocking (the default). Any other value is a requested divisor for the external clock. Although Open Transport serial endpoint providers do not support synchronous communications protocols, you can use this option to select an external timing signal for synchronous clocking between the sender and receiver. Be aware that not all serial implementations support an external clock and that not all requested divisors will be supported if such an implementation does support an external clock.

SERIAL_OPT_BURSTMODE

Requests burst mode operation. A value of 0 turns off burst mode (the default) and a 1 requests burst mode to be turned on. In burst mode, the serial driver continues looping, reading incoming characters, rather than waiting for an interrupt for each character. This option may not be supported by all serial drivers.

▲ **WARNING**
Note that burst mode may adversely impact performance of the Macintosh system, since interrupts may be held off for long periods of time. ▲

Serial-Specific Commands

Serial endpoints support several serial-specific commands that use the `OTIoctl` function, which is described in the chapter "Providers" in this book. This function accesses the equivalent low-level serial driver control or status function (`PBControlAsync` or `PBStatusAsync`). The `csCode` value for each routine is listed in each command's description. For information about Device Manager functions for opening, closing, and communicating with device drivers, see the book *Inside Macintosh: Devices*.

I_SetSerialDTR

This command sets the DTR signal on the serial port. Use the constant `kOTSerialSetDTROff` to turn the DTR signal off, and `kOTSerialSetDTROn` to turn the DTR signal on. The following line of code turns DTR on:

```
OTIoctl(theSerialEndpoint, I_SetSerialDTR, kOTSerialSetDTROn);
```

Asserting the DTR signal is equivalent to using a serial driver control call with a `csCode` value of 1,7 and negating the DTR signal is equivalent to using a `csCode` value of 18.

I_SetSerialBreak

This command controls a break signal on the serial connection. It is a 4-byte unsigned integer. Its value is `kOTSerialSetBreakOff` to unconditionally turn the break signal off, `kOTSerialSetBreakOn` to unconditionally turn the break signal on, and any other value to turn the break signal on for a specified number of milliseconds. The following line of code turns the break on:

```
OTIoctl(theSerialEndpoint, I_SetSerialBreak, kOTSerialSetBreakOn);
```

Asserting a break signal is equivalent to using a serial driver control call with a `csCode` value of 12, and deasserting the break signal is equivalent to using a `csCode` value of 11.

Note
Note that, contrary to some readers' expectations, the on value is 0 and the off value is 1. ◆

I_SetSerialXOffState

This command sets the XOFF state of the serial port. Setting XOFF is equivalent to receiving an XOFF character, and clearing XOFF is equivalent to receiving an XON character. A value of `kOTSerialForceXOffFalse` unconditionally clears the XOFF state, while a value of `kOTSerialForceXOffTrue` unconditionally sets it. The following line of code unconditionally sets the XOFF state:

```
OTIoctl(theSerialEndpoint, I_SetSerialXOffState, kOTSerialForceXOffTrue);
```

Setting the XOFF state is equivalent to using a serial driver control call with a `csCode` value of 21, and clearing the XOFF state is equivalent to using a `csCode` value of 22.

I_SetSerialXOn

This command causes the serial port to send an XON character. A value of `kOTSerialSendXOnIfXOffTrue` causes it to be sent only if the endpoint is in the XOFF state (that is, if the last input flow-control character sent was XOFF), while a value of `kOTSerialSendXOnAlways` unconditionally sends the character. The following line of code unconditionally sends an XON character:

```
OTIoctl(theSerialEndpoint, I_SetSerialXOn, kOTSerialSendXOnAlways);
```

Sending the XON character unconditionally is equivalent to using a serial driver control call with a `csCode` value of 24, and sending the XON character when an endpoint is in an XOFF state is equivalent to using a `csCode` value of 23.

I_SetSerialXOff

This command causes the serial port to send an XOFF character. A value of `kOTSerialSendXOffIfXOnTrue` causes it to be sent only if the endpoint is in the XON state (that is, if the last input flow control character sent was XON), while a value of `kOTSerialSendXOffAlways` unconditionally sends the character. The following line of code unconditionally sends an XOFF character:

```
OTIoctl(theSerialEndpoint, I_SetSerialXOff, kOTSerialSendXOffAlways);
```

Sending the XOFF character unconditionally is equivalent to using a serial driver control call with a `csCode` value of 26, and sending the XOFF character when an endpoint is in an XON state is equivalent to using a `csCode` value of 25.

I_SetFramingType

This command sets the framing type for a serial port. Currently, serial ports can support four different framing types, as enumerated in the `fCapabilities` field of the `OTPortRecord`. These are `kOTSerialFramingAsync`, `kOTSerialFramingHDLC`,

kOTSerialFramingSDLC, and kOTSerialFramingAsyncPackets. The normal mode of operation is kOTSerialFramingAsync. You can change the mode of operation to the asynchronous packet framing type by making this OTIoctl command:

```
OTIoctl(theSerialEndpoint, I_OTSetFramingType, kOTSerialFramingAsyncPackets);
```

If you select the kOTSerialFramingAsyncPackets type, the underlying serial provider assumes that each individual message that arrives is a separate packet, and should be sent as such. It also means that the underlying provider ensures that if data is flushed, all data will be flushed except any packet that is being processed at the time of the flush. This behavior is important to technologies like Apple Remote Access (ARA) or Point-to-Point Protocol (PPP) implementations, which use the serial port for delivery of discrete packets, because

- stopping a packet in the middle of a transfer causes a performance degradation while the upper protocols expend effort to resynchronize
- both protocols want to ensure that if they have to flush the queue of waiting messages that all waiting messages are flushed, even if they are queued up in the protocol module.

Appendixes

Open Transport and XTI

This appendix describes the correspondence between the XTI and Open Transport client programming interfaces. Open Transport is a superset of XTI and therefore includes functions that are not defined in XTI. The XTI interface is not the preferred interface to use in Open Transport applications; however, if you are porting an existing XTI application, the XTI interface provides the simplest migration path.

This appendix describes

- how XTI functions correspond to Open Transport functions and vice versa
- how XTI data structures correspond to Open Transport data structures
- how XTI error codes correspond to Open Transport result codes

The Open Transport interface currently defines functions and data structures for four different kinds of providers. This appendix focuses on how general provider functions and endpoint functions correspond to XTI functions. Because mapper provider functions and service provider functions are an extension to XTI, they are not included in this appendix.

You should read this appendix if you need a simple summary of the differences between the XTI and Open Transport interfaces or if you plan to convert an application using an XTI interface to the preferred C interface.

Open Transport Programming Interfaces

The Open Transport library includes three related client programming interfaces: XTI-style, preferred C, and preferred C++. The XTI-style interface includes the C-language XTI functions, plus some Open Transport extensions. XTI is not the preferred interface for the Macintosh because it handles errors through the use of a global variable. Nevertheless, an XTI interface is provided to ease porting of existing XTI client applications.

IMPORTANT

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 A-1 shows the correspondence between XTI functions and Open Transport functions.

Table A-1 XTI-to-Open Transport function cross-reference

XTI function	Open Transport function
`t_accept`	`OTAccept`
`t_alloc`	`OTAlloc`
`t_bind`	`OTBind`
`t_close`	`OTCloseProvider`
`t_connect`	`OTConnect`
`t_error`	—
`t_free`	`OTFree`
`t_getprotaddr`	`OTGetProtAddress`
`t_getinfo`	`OTGetEndpointInfo`
`t_getstate`	`OTGetEndpointState`
`t_listen`	`OTListen`
`t_look`	`OTLook`
`t_open`	`OTOpenEndpoint`
`t_optmgmt`	`OTOptionManagement`

continued

Table A-1 XTI-to-Open Transport function cross-reference (continued)

XTI function	Open Transport function
t_rcv	OTRcv
t_rcvconnect	OTRcvConnect
t_rcvdis	OTRcvDisconnect
t_rcvrel	OTRcvOrderlyDisconnect
t_rcvudata	OTRcvUData
t_rcvuderr	OTRcvUDErr
t_snd	OTSnd
t_snddis	OTSndDisconnect
t_sndrel	OTSndOrderlyDisconnect
t_sndudata	OTSndUData
t_strerror	—
t_sync	OTSync
t_unbind	OTUnbind

Table A-2 describes shows the correspondence between Open Transport functions and XTI functions.

Table A-2 Open Transport-to-XTI function cross-reference

Open Transport function	XTI function
OTAccept	t_accept
OTAckSends	—
OTAlloc	t_alloc
OTBind	t_bind
OTCloseProvider	t_close
OTConnect	t_connect

continued

Table A-2 Open Transport-to-XTI function cross-reference (continued)

Open Transport function	XTI function
OTDontAckSends	—
OTFree	t_free
OTGetEndpointInfo	t_getinfo
OTGetProtAddress	t_getprotaddr
OTGetNotifier	—
OTGetEndpointState	t_getstate
OTInstallNotifier	—
OTIsNonBlocking	—
OTIsSynchronous	—
OTListen	t_listen
OTLook	t_look
OTOpenEndpoint	t_open
OTOptionManagement	t_optmgmt
OTRcv	t_rcv
OTRcvConnect	t_rcvconnect
OTRcvDisconnect	t_rcvdis
OTRcvOrderlyDisconnect	t_rcvrel
OTRcvRequest	—
OTRcvUData	t_rcvudata
OTRcvUDErr	t_rcvuderr
OTRcvURequest	—
OTRemoveNotifier	—
OTResolveAddress	—
OTSetAsynchronous	—
OTSetBlocking	—

continued

Table A-2 Open Transport-to-XTI function cross-reference (continued)

Open Transport function	XTI function
OTSetNonBlocking	—
OTSetSynchronous	—
OTSnd	t_snd
OTSndDisconnect	t_snddis
OTSndOrderlyDisconnect	t_sndrel
OTSndReply	—
OTSndRequest	—
OTSndUData	t_sndudata
OTSndUReply	—
OTSndURequest	—
OTSync	t_sync
OTUnbind	t_unbind

Extensions to XTI

Table A-3 lists the Open Transport endpoint and general provider functions that are not part of XTI. Although this document refers to these functions by their Open Transport preferred-C names, you can also call these functions by the XTI-style names listed in the table.

Table A-3 Open Transport Functions not found in XTI

Open Transport preferred-C name	XTI-style name
OTAckSends	—
OTDontAckSends	—
OTGetProtAddress	t_getprotaddr
OTInstallNotifier	t_installnotifier
OTIsNonBlocking	t_isnonblocking
OTIsSynchronous	t_issynchronous
OTRcvRequest	t_rcvrequest
OTRcvURequest	t_rcvurequest
OTRemoveNotifier	t_removenotifier
OTResolveAddress	t_resolveaddr
OTSetAsynchronous	t_asynchronous
OTSetSynchronous	t_synchronous
OTSndReply	t_sndreply
OTSndRequest	t_sndrequest
OTSndUReply	t_sndureply
OTSndURequest	t_sndurequest

Data Structures

Many of the Open Transport functions take pointers to data structures as parameters. Table A-4 shows the standard XTI data structure names and the corresponding preferred-C interface structure names.

Table A-4 XTI-to-Open Transport data structure cross-reference

XTI name	Open Transport name
`int fd`	`EndpointRef`
`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 the 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 with defined constants for each; for example, `TBADADDR` or `TFLOW`.

When an Open Transport preferred-C function fails, the error code is returned as the result of the function. Open Transport does not use global variables to

store error results and, to remain consistent with the Macintosh Toolbox, it specifies all errors as negative numbers. Open Transport result codes have names like `kOTBadAddressErr` and `kOTFlowErr`. There is a corresponding Open Transport result code for every XTI result code, as shown in Table A-5. For an explanation of Open Transport result codes, see Appendix B in this book.

Table A-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

continued

Table A-5 XTI-to-Open Transport result code cross-reference (continued)

XTI result code	Open Transport result code
TNOSTRUCTYPE	kOTStructureTypeErr
TNOTSUPPORT	kOTNotSupportedErr
TNOUDERR	kOTNoUDErrErr
TOUTSTATE	kOTOutStateErr
TPROTO	–
TPROVMISMATCH	kOTProviderMismatchErr
TQFULL	kOTQFullErr
TRESADDR	kOTResAddressErr
TRESQLEN	kOTResQLenErr
TSTATECHNG	kOTStateChangeErr
TSYSERR	-

Result Codes

This appendix lists the result codes that Open Transport preferred-C functions return, as shown in Table B-1. For information about XTI result codes, refer to the X/Open Transport Interface specification.

Table B-1 Open Transport result codes

Result code	Value	Meaning
`kOTNoError`	0000	The function completed execution without error.
`kOTBadAddressErr`	–3150	The specified protocol address was in an incorrect format or contained illegal information. For TCP/IP this means that the address does not exist in the specified domain.
`kOTBadOptionErr`	–3151	The specified protocol options were in an incorrect format or contained illegal information.
`kOTAccessErr`	–3152	You do not have permission to negotiate the specified address or options.
`kOTBadReferenceErr`	–3153	The specified endpoint reference does not refer to a valid endpoint.
`kOTNoAddressErr`	–3154	You failed to supply an address, or the endpoint could not allocate an address.
`kOTOutStateErr`	–3155	The endpoint was not in an appropriate state when you called this function.
`kOTBadSequenceErr`	–3156	You specified an invalid sequence number or a `NULL` pointer for the `call` parameter when rejecting a connection request.

continued

Table B-1 Open Transport result codes

Result code	Value	Meaning
kOTLookErr	–3158	An asynchronous event has occurred. If the event has occurred for an endpoint, you can use the OTLook function to find out what event it was or you can use the notifier function if your notifier handles asynchronous events. If the event has occurred for a provider other than an endpoint, the notifier function installed for that provider must handle the asynchronous event .
kOTBadDataErr	–3159	The amount of client data you specified was not within the bounds allowed by the endpoint.
kOTBufferOverflowErr	–3160	The buffer you allocated to store information when this function returns is not sufficiently large to store the incoming data.
kOTFlowErr	–3161	The endpoint is in asynchronous mode, but the flow-control mechanism prevents the endpoint from accepting or sending any data at this time.
kOTNoDataErr	–3162	For an endpoint or mapper, this result is returned when the endpoint is in nonblocking mode, but no data is currently available. For a mapper, this result is returned by the OTLookupName function when no names are found. TCP/IP functions return this result when no data is available, a lookup times out, or a name exists but addresses do not.
kOTNoDisconnectErr	–3163	No disconnection indication is available.
kOTNoUDErrErr	–3164	No unit data error indication currently exists on this endpoint.
kOTBadFlagErr	–3165	You specified an invalid flag value.
kOTNoReleaseErr	–3166	No orderly release indication currently exists on this endpoint.
kOTNotSupportedErr	–3167	This action is not supported by this endpoint.

continued

Table B-1 Open Transport result codes

Result code	Value	Meaning
kOTStateChangeErr	–3168	The endpoint is undergoing a transient state change. This error is returned when you call a function while an endpoint is in the process of changing states. You 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.) The provider also returns this error if you attempt to call an "incompatible" function while another operation is still ongoing; for example if you call the function OTSndUData while a call to the OTOptionManagement function is still outstanding.
kOTStructureTypeErr	–3169	You specified an unsupported structure type for the structType parameter of the OTAlloc function. This error is also returned when the structType structure you specify is inconsistent with the endpoint type.
kOTBadNameErr	–3170	You specified an invalid endpoint name. This error is returned by the TCP/IP domain name resolver (DNR) if you specify a bad host name.
kOTBadQLenErr	–3171	You are using this endpoint to listen for connection requests, but when you bound the endpoint, you specified 0 for the qlen field. If you want to use an endpoint to listen for connection requests, the value of the qlen field must be greater than 0.
kOTAddressBusyErr	–3172	As a return value for a call to the OTBind function, this error code indicates one of the following conditions: 1) no dynamic addresses are available for protocols or configuration methods that allow dynamic addressing, 2) you are attempting to bind two connectionless endpoints to the same address, or 3) you are attempting to bind two or more connection-oriented endpoints to the same address and more than one of these endpoints has a qlen field greater than 0.

continued

Table B-1 Open Transport result codes

Result code	Value	Meaning
kOTIndOutErr	–3173	There are outstanding connection indications on the endpoint, and you are accepting a connection for this endpoint. When accepting a connection for an endpoint that is listening for connection requests, you must have responded to all outstanding requests either by rejecting them with the OTSndDisconnect function or by accepting them with the OTAccept function.
kOTProviderMismatchErr	–3174	The endpoint that is to accept the connection is not the same kind of endpoint as the endpoint listening for the connection. The listening and accepting endpoints must be the same kind: either connection-oriented transaction-based or connection-oriented transactionless.
kOTResQLenErr	–3175	When this endpoint was bound, the qlen field was set to a value greater than 0. But to accept a connection on an alternate endpoint that is bound to the same address, such as this one, the endpoint must be bound with a qlen parameter equal to 0.
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.
kOTQFullErr	–3177	The maximum number of outstanding indications, as specified by the value of the qlen field you used when you bound the endpoint, has been reached for the endpoint.
kOTProtocolErr	–3178	An unspecified protocol error occurred.
kOTBadSyncErr	–3179	You called the OTSync function at non-System Task time.
kOTCanceledErr	–3180	A provider function never finished executing because the provider was closed or because the function was synchronous and synchronous functions were cancelled.

continued

Table B-1 Open Transport result codes

Result code	Value	Meaning
kOTNotFoundErr	–3201	Requested information does not exist.
kENOENTErr	–3201	This error literally means no such file or directory. In XTI (and Open Transport), a function returns this result when you try to open an endpoint or mapper that does not exist in the system.
kENIOErr	–3204	An I/O error occurred.
kENXIOErr	–3205	No such device or address.
kEAGAINErr	–3210	The provider cannot perform this operation at this time. Try again later.
kENOMEMErr	–3211	Open Transport cannot allocate enough memory to meet your request.
kOTOutOfMemoryErr	–3211	Open Transport has run out of internal memory. This might happen, for example, if you are doing a lot of asynchronous sends and not acknowledging sends, which means that Open Transport has to copy the data being sent into its own internal buffers.
kEBUSYErr	–3215	The device you are trying to access is busy and could not complete your request.
kOTDuplicateFoundErr	–3216	You are attempting to register a port or other entity that already exists.
kEINVALErr	–3221	The specified address had an invalid size.
kEWOULDBLOCKErr	–3234	In order to complete the requested operation, the endpoint provider would have to block, and the endpoint is in nonblocking mode.
kETIMEDOUTErr	–3259	The requested operation timed out.
kENOSRErr	–3271	Open Transport cannot allocate enough system resources (usually stream messages) to meet your request.

Glossary

abortive disconnect A type of disconnection that breaks a connection without the knowledge of the remote peer. An abortive disconnect can result in loss of data. See also **orderly disconnect.**

absolute requirement A type of option that a protocol implementation can neither ignore nor negotiate to a partly successful value.

active peer An endpoint provider that initiates connection requests. The use of an active peer is typical of a client-server environment in which an endpoint, the active peer, attempts to establish a connection with a passive peer, such as a file server, that listens for connection requests. See also **passive peer.**

address type An AppleTalk address attribute that identifies the type of address format used for an AppleTalk endpoint.

ADSP AppleTalk Data Stream Protocol.

AEP Echoer A DDP client process that implements the AppleTalk Echo Protocol (AEP).

at-least-once transaction A type of transaction that ensures that an ATP responder receives every request directed to it at least once. These transactions are also referred to as *ALO transactions*. See also **exactly-once transactions.**

AppleTalk Echo Protocol (AEP) An AppleTalk protocol that is a client of DDP. This protocol can measure the performance of an AppleTalk network and test for the presence of a given node.

AppleTalk internet A number of interconnected AppleTalk networks. An AppleTalk internet can include a mix of LocalTalk, TokenTalk, EtherTalk, and FDDITalk networks, or it can consist of multiple networks of a single type, such as several LocalTalk networks. An AppleTalk internet can include both nonextended and extended networks. See also **internet.** Compare with **Worldwide Internet.**

AppleTalk Session Protocol (ASP) A connection-oriented transaction-based AppleTalk protocol that sets up and maintains sessions between workstations and servers.

AppleTalk service provider An Open Transport provider that gives applications access to information and services that are specific to the AppleTalk protocol stack. Applications use an AppleTalk service provider to obtain zone names and to get information about the current AppleTalk environment for a given machine.

AppleTalk Data Stream Protocol (ADSP) A connection-oriented transactionless AppleTalk protocol that supports sessions over which applications can exchange full-duplex streams of data. In

addition to ensuring reliable delivery of data, ADSP provides a peer-to-peer connection. ADSP also provides an application with a means of sending expedited attention messages.

AppleTalk Secure Data Stream Protocol (ASDSP) The authentication and encryption features of ADSP.

AppleTalk Transaction Protocol (ATP) A connectionless transaction-based AppleTalk protocol that allows two endpoints to execute request-and-response transactions. Either ATP endpoint can request another ATP endpoint to perform an action; the other ATP endpoint then carries out the action and transmits a response reporting the outcome.

AppleTalk Transition Queue (ATQ) In classic AppleTalk, the AppleTalk Transition Queue (ATQ) informs an applications each time certain network-related events occur, such as opening or closing an AppleTalk driver. Any applications that rely on the ATQ must use AppleTalk backward compatibility to handle them in the classic AppleTalk manner. See also **miscellaneous events.**

application layer The highest layer of the OSI model. This layer allows for the development of application software. Software written at this layer benefits from the services of all the underlying layers.

ASDSP AppleTalk Secure Data Stream Protocol.

association-related options Options that are tied to a particular connection, transaction, or data transmission; some of the information they contain is destined for the remote client. Compare with **non-association-related options.**

asymmetrical connection A networking connection in which both ends do not have equal control over the communication. A transaction-based connection is an asymmetrical connection. Compare with **symmetrical connection.**

asynchronous communication **A** way of coordinating serial data transfers that is the prevailing standard in the personal computer industry. This method uses a timing scope of a single byte to differentiate bits in the data stream.

asynchronous event An event used to notify your application that something requires immediate attention. For example, expedited data has arrived or a disconnection request is pending. See also **provider event, notifier function.**

asynchronous mode A mode of operation in which provider functions return as soon as they are queued for execution. When the function actually finishes executing, the provider issues a completion event. Compare with **synchronous mode.**

ATP AppleTalk Transaction Protocol.

baud rate The rate in samples per second at which a serial receiver samples a line's voltage level.

best-effort delivery A message-delivery paradigm in which the networking protocol attempts to deliver any packets that meet certain requirements, such as containing a valid destination address, but the protocol does not inform the sender when it is

unable to deliver the data, nor does it attempt to recover from error conditions and data loss. Compare with **reliable delivery.**

binding The process of associating an endpoint with a logical address before the endpoint can be used to transfer data.Depending on the protocol you use, you can specify this address as a symbolic name or as a network address. Specific address binding rules and address formats also vary with the protocol you use.

bit time The periodic interval at which a serial receiver samples a line's voltage level.

blocking A mode of operation in which a provider must wait for some action to complete before continuing operation when sending or receiving data. If a provider is blocking, any function used to send or receive data does not return until it has actually completed the operation, even if it has to wait indefinitely.

blocking status A provider's state that determines whether it is blocking. See also **blocking, nonblocking.**

break signal **A** special signal that falls outside the character frame. The break signal occurs when the line is switched from the mark state to a space and held there for longer than a character frame.

bridge A device that connects networking cables without examining the addresses of messages or making decisions as to the best route for a message to take. Compare with **router**, **gateway**.

canonical name A fully qualified domain name that is not an alias.

character frame The unit of serial communication transmission. Character frames of 7 or 8 data bits are commonly used for transmitting ASCII characters.

child port An attribute of a port that identifies which of multiple available ports a pseudodevice uses as its transmission hardware. A port may have more than one child port, all of which can be active simultaneously.

classic AppleTalk The implementation of AppleTalk available before Open Transport.

Clear To Send (CTS) signal **A** signal that indicates that the modem or printer is ready to send data. Since most communications between microcomputers are full-duplex nowadays, the CTS signal is permanently asserted.

client A protocol that uses the services of an underlying protocol. For example, ADSP is a client of DDP.

combined DDP-NBP address format An AppleTalk address format that combines an endpoint's physical address and its NBP name. See also **DDP address format, NBP address format.**

completion event A provider event used to notify your application that an asynchronous function has completed execution. See also **provider event, notifier function.**

connection An association between two endpoints that permits the establishment and maintenance of an exclusive dialogue between the endpoints.

connectionless protocol A networking protocol in which a node that wants to communicate with another simply sends a message without first establishing that the receiving node is prepared to receive it. Each message sent must include addressing information so that it can be delivered to its destination. Compare with **connection-oriented protocol.**

connection-oriented protocol A networking protocol in which two nodes on the network that want to communicate first establish a connection. Once a connection is established, the communicating applications or processes on the nodes at either end can send and receive data without having to add addresses to the messages or repeat the handshake process. Compare with **connectionless protocol**. See also **connection, handshake, session.**

datagram A small unit of data that includes a header portion that holds the destination address (and may contain other information, such as a checksum value), and a data portion that holds the message text. Same as **packet.**

Datagram Delivery Protocol (DDP) A connectionless transactionless AppleTalk protocol that transfers data between sockets as discrete datagrams, each carrying its destination socket address. DDP provides best-effort delivery of data.

data-link layer The layer of the OSI model that, together with the physical layer, provides for connectivity. The data-link layer contains the software that communicates directly with the physical network devices and provides for switching between physical devices.

DDP Datagram Delivery Protocol.

DDP address format An AppleTalk address format that indicates the physical address of an endpoint. See also **combined DDP-NBP address.**

DDP type The type of protocol. This is used by DDP endpoints to filter incoming and outgoing data.

Data Terminal Reedy (DTR) signal A signal indicates that the computer is ready to communicate. Deasserting this signal causes the modem or printer to suspend transmission.

default port The port that Open Transport uses when a specific port is not indicated. For example, the LocalTalk default port is the printer port, “ltlkB.

domain A collection of hosts on a TCP/IP internet. Domains are hierarchically arranged and each can be identified by its domain name or its IP address.

domain name A character-string name that can be used to identify a TCP/IP domain. See also **fully qualified domain name.**

domain name resolver A process running on a TCP/IP network that translates between the character-string names used by people to identify nodes on the internet and the 32-bit internet addresses used by the network itself.

dynamically assigned socket An AppleTalk socket arbitrarily assigned by DDP if you do not specify a socket number when binding an endpoint. Compare **statically assigned socket.**

echo request packet A packet sent by the AEP Echoer to the target node.

echo reply packet The packet sent in response to an echo request packet sent by the AEP Echoer.

echoer socket The statically assigned socket (socket number 4) that AEP uses to listen for echo packets.

endpoint The communications path between your application and an endpoint provider. An endpoint consists of a set of data structures that are maintained by Open Transport and that specify the components of the endpoint provider, the provider's state, and the provider's mode of operation.

endpoint function An Open Transport function that you can use only with endpoints. Endpoint functions create and bind endpoints, obtain information about endpoints, establish and break down connections, and transfer data. The behavior of an endpoint function is determined by the endpoint's mode of operation.

endpoint provider An Open Transport provider that sends and receives information over a data link. See also **endpoint, mapper provider, service provider.**

endpoint reference A number that Open Transport returns to you when you open an endpoint. This number identifies the instance of the endpoint provider that you have created.

endpoint state An endpoint attribute that governs which endpoint functions you can call for the endpoint. For example, a connectionless endpoint can only transfer data while it is in the `T_IDLE` state; a connection-oriented endpoint can only transfer data while it is in the `T_DATAXFER` state.

ETSDU See **expedited transport service data unit.**

event See **provider event.**

exactly-once transaction An ATP transaction that ensures that the responder receives a specific request only once. These are also referred to as *XO transactions*.

expedited transport service data unit (ETSDU) A unit of expedited data that you can use to deliver urgent data. An ETSDU is the largest piece of expedited data that an endpoint can transfer with boundaries and content preserved. Different types of endpoints permit different size ETSDUs. See also **transport service data unit (TSDU).**

extended network An AppleTalk network that has a range of network numbers assigned to it and that supports multiple zones. Each node on the network has a unique network number-node ID combination to identify it.

full duplex A networking connection in which both ends can transmit and receive data simultaneously. Compare with **half duplex.**

fully qualified domain name A domain name that corresponds to an internet address.

gateway A device that connects networking cables and that converts addresses and protocols to connect dissimilar networks. Compare with **bridge, router.**

general provider function A function that you can use to manipulate any type of provider. For example, you can call the `OTCloseProvider` function to close any type of provider. See also **provider function.**

half duplex A networking connection in which the two ends have to take turns transmitting and receiving. Compare with **full duplex.**

handshake A connection-establishment process involving the exchange of predetermined signals between nodes in which each end identifies itself to the other. See also **connection-oriented protocol, session.**

header The portion of a datagram that holds the destination address and may contain other information, such as a checksum value.

host A node on a TCP/IP internet. A host that is addressable by other hosts has a host name and one or more domain names.

internet A set of networks connected by routers or gateways.

Internet See **Worldwide Internet.**

internet address A 32-bit number that uniquely identifies a host on a TCP/IP network. An internet address is commonly expressed in dotted-decimal notation (for example, "12.13.14.15") or hexadecimal notation (for example, "0x0c0d0e0f"). Also called **IP address.**

Internet Protocol (IP) A basic datagram-delivery protocol; part of the TCP/IP protocol family.

IP Internet Protocol.

IP address An internet address.

mail exchange Any TCP/IP host that can accept mail for another host or for a domain. A mail exchange can be a mail server, a router, or just a host configured to accept and pass on mail.

mail preference value A number used by a mail application to determine to which mail exchange to deliver a message when there is more than one that can accept mail for a particular domain. The mailer sends the mail to the mail exchange with the lowest preference value first and tries the others in turn until the mail is delivered or until the mailer deems the mail undeliverable.

mapper The communications path between your application and a mapper provider. A mapper consists of a set of data structures, maintained by Open Transport, that specify the components of the mapper provider, the provider's state, and the provider's mode of operation.

mapper provider An Open Transport provider that relates network addresses to network node names and can be used to register and remove node names for networks that support this ability. See also **endpoint provider, mapper, service provider.**

mapper reference A number that Open Transport returns to you when you open a mapper. This number identifies the instance of the mapper provider that you have created.

mark state An idle state in which a serial communications line has a positive voltage.

miscellaneous event A network-related event that may affect the operation of an Open Transport provider. In particular, these apply to AppleTalk endpoints and can include such events as opening or closing an AppleTalk driver. See also **AppleTalk Transition Queue (ATQ).**

mode of operation An endpoint attributes that determines whether provider functions execute synchronously or asynchronously, whether functions can wait to send or receive data, and whether your application is notified when a function has sent data.

module name A port structure field that gives the name of the actual Streams module that implements the driver for a given port. Open Transport uses this name internally.

multihoming The situation in which a single host or node is connected to two or more networks or network interface cards (NICs) at the same time.

multinode An node ID that an application can acquire that is in addition to the standard node ID that is assigned when the node joins an AppleTalk network.

multinode address format An AppleTalk address format that indicates the physical address of a multinode endpoint.

multinode architecture An AppleTalk feature that allows an application to acquire node IDs that are additional to the standard node ID that is assigned to the system when the node joins an AppleTalk network.

multinode ID An identifier that allows the computer running your application to appear as multiple nodes on the network even though it is only one physical entity. Each acquired multinode is in addition to the standard node ID already assigned to the computer when it joined the network as a node.The prime example of a multinode application is Apple Remote Access (ARA).

multiport identifier A port function parameter that distinguishes between multiple ports when a single slot supports more than one port. Typically, the hardware device in a multiport slot is either a plug-in multifunction card with multiple ports on it or a device with multiple uses, one or more of which is a port.

Name-Binding Protocol (NBP) An AppleTalk protocol that maintains a mapping of logical names (like those in the Chooser) to physical socket addresses in such a way that if the node ID changes, you can continue to reliably identify your application.

NBP Name Binding Protocol.

name That part of an NBP name that typically identifies the user of the system or, in the case of a server, the system itself.

NBP address format An AppleTalk address format that indicates the endpoint's NBP name.

NBP entity structure A structure that Open Transport provides for convenient manipulation of NBP names. The NBP entity structure itself does not contain escape characters, but the NBP entity extraction functions insert a backslash (\) in front of any backslash, colon (:), or at sign (@) they find in an NBP name so that mapper functions can use a correctly formatted NBP name.

NBP mapper provider An Open Transport mapper provider that is configured as an NBP mapper.

NBP name An endpoint's logical name, sometimes called its *entity name,* used in the NBP address format. The NBP name consists of three fields: name, type, and zone. See also **name, type, zone.**

network A system of computers and other devices (such as printers and modems) that are connected in such a way that they can exchange data.

network layer The layer of the OSI model immediately above the data-link layer. The network layer specifies the network routing of data packets between nodes and the communications between networks, which is referred to as *internetworking.*

node An addressable physical device connected to a network. See also **node ID.**

node ID An 8-bit number that identifies a node.

noise Environmental perturbations that can affect an electrical line. Noise can cause errors in transmission by altering voltage levels so that a bit is reversed, shortened, or lengthened.

non-association–related options Options that are negotiated between the client and its endpoint provider. Such options contain no information for the remote client. See also Compare with **association-related options.**

nonblocking A mode of operation in which a provider cannot wait when sending or receiving data. If a provider is nonblocking, any provider function used to send or receive data returns with an error result if it cannot complete the operation immediately. Compare with **blocking.**

nonextended network An AppleTalk network that has one network number assigned to it and that supports only one zone. On nonextended networks, all nodes share the same network number and zone name, and each node has a unique node ID. Compare with **extended network.**

notifier function A callback function that handles Open Transport provider events. See also **provider event.**

option A value you can set for an endpoint that is of interest to a specific protocol. For example, an option might enable or disable checksums or specify the priority of a datagram. The available options and their significance are defined by each implementation of each protocol. Every option has a default value.

option negotiation The process of trying to replace one or more default option values with other values. A negotiation might involve of a client and its endpoint provider, or both a local and remote client and their endpoint providers. A successful negotiation results in obtaining exactly the

option values requested, a partly successful negotiation results in getting different values for the options requested, and a failed negotiation results in not being able to change existing values at all. See also **absolute requirement.**

orderly disconnect Breaking a connection with the knowledge and cooperation of the remote peer. This method of disconnection prevents loss of data. Orderly disconnects can be either remote (over-the-wire) disconnects or local disconnects. See also **abortive disconnect.**

OSI model A standard reference model for network architectures. The OSI (Open Standards Interconnection) model describes a seven-layer structure for networking protocols. See also **application layer, presentation layer, session layer, transport layer, network layer, datalink layer, physical layer.**

packet A small unit of data that includes a header portion that holds the destination address (and may contain other information, such as a checksum value), and a data portion that holds the message text. Same as **datagram.**

PAP Printer Access Protocol.

passive peer An endpoint provider that listens for incoming connection requests. The use of a passive peer is typical of a server environment in which a server, such as a file server, uses an endpoint to listen for connection requests from multiple remote endpoints. Endpoints throughout the network can contact the server's passive endpoint with connection requests. See also **active peer.**

peer-to-peer connection See **symmetrical connection.**

physical layer The layer of the OSI model that provides for physical connectivity. The communication between networked systems can be via a physical cable made of wire or optical fiber, or it can be via infrared or microwave transmission. In addition to these, the hardware can include a network interface controller (NIC), if one is used. The physical layer includes the network hardware and the drivers that control the hardware.

port A logical entity that combines a hardware device and the software driver that acts as an interface to it. Ethernet, serial devices, and LocalTalk ports are examples of ports commonly used in Open Transport. See also **child port, default port, multiport identifier, pseudodevice.**

port alias A port structure flag that identifies the default port for LocalTalk. The port alias is a port name of "ltlk." Because it has the same Streams module name as the default LocalTalk port, if you use the port alias in the configuration string, Open Transport can locate the default port even in those cases where a computer doesn't use the standard default of "ltlkB." See also **default port.**

port name A unique name that designates the port. It is typically an abbreviation of the port's device type plus a suffix, usually numeric, such as "enet0," "enet1," and "enet2." For historic reasons, LocalTalk and serial ports use an alphabetic suffix instead.

port reference A 32-bit value that uniquely describes a port's hardware characteristics: its device and bus type, its physical slot number, and, where applicable, its multiport identifier.

port registry An Open Transport registry of available ports.

port structure A structure that uniquely identifies each port on a system. A port structure contains each port's port reference, several sets of information flags, its port name, its Streams module name, and the slot ID (for ports on a PCI bus).

presentation layer The layer of the OSI model immediately below the application layer. Protocols in this layer assume that an end-to-end path or connection already exists across the network between the two communicating parties. Protocols in this layer are concerned with the representation of data values for transfer, or the transfer syntax.

Printer Access Protocol (PAP) An asymmetrical connection-oriented transactionless AppleTalk protocol that enables communication between client and server endpoints, allowing multiple connections at both ends. In particular, PAP is used for direct printing to AppleTalk printers.

privileged options Options whose value can be changed only by privileged clients, although it is sometimes possible for nonprivileged clients to read the value of a privileged option.

protocol option See **option.**

protocol stack A set of protocols related in a hierarchical fashion, where the higher-level protocols are clients of the lower-level protocols.

provider A layered set of Streams modules and drivers that provides a service to clients of Open Transport. See also **endpoint provider, mapper provider, service provider.**

provider event An event Open Transport uses to notify your application that something has occurred that demands immediate attention or that an asynchronous function has completed execution. See also **asynchronous event** and **completion event.**

provider function A function that you can call to manipulate a specific type of provider. For example, you call the `OTOpenEndpoint` function to open an endpoint provider. See also **general provider function.**

provider reference A value that is returned to you when you open a provider and that you must pass back when you call a provider function. The data type of the provider reference depends on the type of the provider.

pseudodevice A special type of port that is a driver that doesn't interface to a hardware device; instead, it interfaces to other device drivers. A pseudodevice uses a special device type, designated with the constant `kOTPseudoDevice`, and each must have a unique port reference. See also **child port.**

RawIP An application interface to the IP protocol.

read-only options Options whose value you can read but not change.

receive queue A receiving buffer used to store incoming data until the local endpoint provider acknowledges reading it.

reliable delivery A message-delivery paradigm in which the networking protocol includes error checking and recovery from error or loss of data. Compare with **best-effort delivery.**

requester An endpoint that as part of a transaction sends a request for a service. The responder endpoint reads the request, performs the service, and sends a reply. When the requester receives the reply, the transaction is complete.

responder An endpoint that as part of a transaction reads a requester endpoint's request, performs the service, and sends a reply.

router A device that connects networking cables and that contains addressing and routing information that lets it determine from a message's address the most efficient route for the message. A message can be passed from router to router several times before being delivered to its destination. Compare with **bridge, gateway.**

send-acknowledgment status A provider's attribute that determines whether endpoint providers that send data make an internal copy of the data before sending it and whether they notify your application when they have sent the data.

send queue A buffer used to store outgoing data until the remote endpoint provider acknowledges receiving it.

service provider An Open Transport provider that handles features unique to a specific type of Open Transport service. For example, to get information about AppleTalk zones, you must open an AppleTalk service provider. See also **endpoint provider, mapper provider.**

session A logical (as opposed to physical) connection between two entities on a network or internet. A session must be set up at the beginning, maintained by the periodic exchange of information, and broken down at the end. See also **connection-oriented protocol.**

session layer The layer of the OSI model that serves as an interface to the transport layer, which is below it. The session layer allows for establishing a session, which is the process of setting up a connection over which a dialog between two applications or processes can occur. Some of the functions that the session layer provides for are flow control, establishment of synchronization points for checks and recovery during file transfer, full-duplex and half-duplex dialogs between processes, and aborts and restarts.

socket A piece of software that serves as an addressable entity on a node. Endpoints exchange data with each other across an AppleTalk internet through sockets.

socket number An 8-bit number that identifies an AppleTalk socket. Each endpoint on an AppleTalk network is associated with a unique 8-bit socket number.

space The state into which a serial line is placed when its voltage is shifted to a negative value.

start bit A signal that delineates a serial line's change from the mark state to a space. The start bit triggers the synchronization necessary for asynchronous communication.

state dependence A condition of a networking protocol or connection in which the response to a request is dependent on a previous request. For example, before a workstation application connected to a file server can read a file, it must have first issued a request to open the file.

statically assigned socket An AppleTalk socket that is permanently reserved for a designated protocol or process. For example, socket 4 is always reserved as the echo socket, used for echoing packets across a network. Compare **dynamically assigned socket.**

stop bit A signal that delineates the end of the character frame and places the serial line back into a positive voltage mark state for a minimum specified time interval. This interval has one of several possible values: 1, 1.5, and 2 stop bits.

Streams module A module that conforms to the Streams architecture. The Streams architecture is a UNIX® standard in which protocols and other service providers are implemented as software modules that communicate between each other using messages. Open Transport software modules are implemented as Streams modules.

subnet A portion of a network, which is in turn a portion of an internet.

subnet mask A number that can be used to determine what portion of an IP address is dedicated to the host identifier and what portion identifies the subnet.

symmetrical connection A networking connection in which both ends have equal control over the communication. Both ends can send and receive data and initiate or terminate the session. Compare with **asymmetrical connection.**

synchronous mode A mode of operation in which provider functions do not return until they have finished executing. See also **asynchronous mode.**

system registry A register of hardware and software configuration information for Power Macintosh computers. The system registry is sometimes referred to as the *Name Registry.*

TCP Transmission Control Protocol.

TCP/IP protocol family A set of networking protocols in wide use throughout the world for government and business applications. The TCP/IP protocol family includes TCP, UDP, and IP, among other protocols.

TCP/IP service provider An Open Transport provider that provides an interface to the TCP/IP Domain Name Resolver (DNR) for clients of Open Transport.

transaction A process during which one endpoint, the *requester,* sends a request for a service. The remote endpoint, called the *responder,* reads the request, performs the

service, and sends a reply. When the requester receives the reply, the transaction is complete.

transaction-based protocol A networking protocol that specifies the sequence and some of the content of messages passed between nodes. Compare with **transactionless protocol.**

transaction ID A number that uniquely identifies a transaction.

transactionless protocol A networking protocol that defines how the data is to be organized and delivered from one node to another but does not specify the sequence or content of messages. Compare with **transaction-based protocol.**

Transmission Control Protocol (TCP) A connection-oriented data stream protocol that provides highly reliable data delivery; part of the TCP/IP protocol family.

transport independence The independence of networking APIs from the underlying networking or transport technology.

transport layer The layer of the OSI model that isolates some of the physical and functional aspects of a network from the upper three layers. It provides for end-to-end accountability, ensuring that all packets of data sent across the network are received and in the correct order. This process involves providing a means of identifying packet loss and supplying a retransmission mechanism. The transport layer may also provide connection and session management services.

transport service data unit (TSDU) A unit of data that allows an endpoint to separate a data stream into discrete logical units when sending and receiving data across a connection. A TSDU is the largest piece of data that an endpoint can transfer with boundaries and content preserved. Different types of endpoints and different endpoint implementations support different size TSDUs. See also **expedited transport service data unit (ETSDU).**

type That part of an NBP name that generally identifies the type of service that the entity provides, for example, "Mailbox" for an electronic mailbox on a server.

UDP User Datagram Protocol.

User Datagram Protocol (UDP) A connectionless datagram protocol that segments data to handle larger datagrams than those allowed by IP; part of the TCP/IP protocol family.

Worldwide Internet The largest worldwide system of networks.

ZIP Zone Information Protocol.

zone A logical grouping of nodes in an AppleTalk network or internet or that part of an NBP name that identifies the zone within the network to which the node belongs.

Zone Information Protocol (ZIP) An AppleTalk protocol that maps network numbers to zone names for all networks belonging to an AppleTalk internet.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Sanborn Hodgkins, Joanna Bujes, Paul Black

LEAD WRITER
Paul Black

DEVELOPMENTAL EDITORS
George Truett, Sanborn Hodgkins

ILLUSTRATOR
Bruce Lee

PRODUCTION EDITOR
Gerri Gray

PROJECT MANAGER
Trish Eastman

Special thanks to Don Coolidge, Marcus Jordan, Rich Kubota, Eric Ockholm, Mike Quinn, Lauren Sherman, Steve Ussery, Tony Wingo

Acknowledgments to Jose Carreon, Michel Guittet, Garry Hornbuckle, Jeri Sonnenberg, and the entire Open Transport engineering team.