INSIDE MACINTOSH

Networking With Open Transport

Revised for version 1.3

November 1997

The Apple logo is a trademark of Apple Computer, Inc.
Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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

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

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

Apple, the Apple logo, Macintosh, AppleTalk, EtherTalk, ImageWriter,LaserWriter, LocalTalk, Mac, MPW, and TokenTalk are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

NuBus is a trademark of Texas Instruments.

PowerPC is a trademark of International Busines Machines Corporation, used under license therefrom.

UNIX is a regisered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Chapter 2 Getting Started With Open Transport 29

Chapter 17 AppleTalk Transaction Protocol (ATP) 323

Chapter 18 Printer Access Protocol (PAP) 333

Chapter 27 Utilities Reference 617

Chapter 28 Advanced Topics Reference 671

Chapter 29 TCP/IP Services Reference 681

Chapter 30 AppleTalk Reference 719

Figures, Tables, and Listings

About This Book

This book, *Inside Macintosh: Networking With Open Transport,* describes the 1.2 release of Open Transport, a communications architecture for implementing network protocols and other communication systems on Mac OS computers. Open Transport provides a set of programming interfaces for applications and other software running on Mac OS computers. This book is about client programming only; it does not include information on how to implement Open Transport network protocol modules or device drivers. That information is covered in the documents provided with the Protocol and Module SDKs.

This book is divided into two parts: the first part provides a conceptual description of Open Transport and instructional examples of how to use it. The second part provides reference information. 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 first chapter "Introduction to Open Transport," defines 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 "Getting Started With Open Transport" is an introductory walk-through a very simple Open Transport program that downloads a URL.

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 chapter "Programming With Open Transport" talks about the structure of Open Transport programs and about how Mac OS interrupt levels affect program execution. The next four chapters, "Option Management," "Ports," "Utilities," and "Advanced Topics," focus on more specialized topics in Open Transport.

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

"Open Transport Reference," the second part of this book, contains complete reference information about the Open Transport API, divided into chapters that correspond to the preceding conceptual ones.

At the end of this book are four appendixes: "Open Transport and XTI," "Result Codes," "Special Functions," and "XTI Option Summary."

- "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.
- "Special Functions." This appendix lists the functions that are callable at hardware interrupt time, the functions that are callable at deferred task time, and the functions that allocate memory.
- "XTI Option Summary" describes option types and option negotiation rules.

For More Information

If you are new to programming for the Macintosh, you can read the book *Inside Macintosh:Overview* for an 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 runtime environment of Macintosh applications, as described in *Mac OS Runtime Architectures* and *Inside Macintosh: Processes.* These and other documents published by Apple Computer may be found at

<http://devworld.apple.com>

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 internet standards specifications, see

<ftp://ds.internic.net/dtd/>

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*.

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.
3 Infinite Loop, M/S 303-2T
Cupertino, CA 95014-6299

<devsupport@apple.com>

Format of a Typical Chapter

Most of the conceptual chapters in this book (the chapters in the first part) 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 explains how to use endpoint functions to transfer data.

The reference chapters in this book (the chapters in the second part) provide complete reference information about the Open Transport API. For example, the chapter "Endpoints Reference" contains these sections:

- "Constants and Data Types." This section includes the constants, types, enumerations, and structures specific to the use of endpoints.

- "Functions." This section presents detailed descriptions of the functions that are used with endpoints.

The reference chapters for the AppleTalk and TCP/IP protocols describe functions and option information that are specific to each protocol.

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. ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. ▲

▲ **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. ▲

The Development Environment

The Open Transport functions described in this book are available using C, C++, or Pascal 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.

PART ONE

Open Transport Essentials

PART ONE

Introduction to Open Transport

Contents

This chapter provides an overview of the 1.2 release of the Open Transport, a communications architecture for implementing network protocols and other communications systems. This book discusses only the implementation of Open Transport on the Mac OS—that is, the set of programming interfaces for applications and other software running on Mac OS 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 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 Mac OS computers. Whereas AppleTalk provided a proprietary networking system for Macintosh computers, the current Macintosh OS 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 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. This feature is described in more detail in "Transport Independence" (page 22).

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 computer at the same time. This feature is currently available only for AppleTalk protocols. **Single link multihoming**, introduced with Open Transport version 1.3, supports multiple IP addresses on the same hardware interface. This feature is available only to TCP/IP protocols.
- **Multinode** support is an AppleTalk feature that allows an application to acquire node IDs in addition to the standard node ID that is assigned to the system when the node joins an AppleTalk network. The prime example of a multinode application is Apple Remote Access (ARA). The chapters "AppleTalk Addressing" (page 279) and "Datagram Delivery Protocol (DDP)" (page 303) 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 .

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" (page 11).

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. In this case, 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 connectionless transactionless protocol sends data in discrete datagrams. A **datagram,** also referred to as a **packet,** is a 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.

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.

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 transactionless AppleTalk Printer Access Protocol (PAP) 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 PAP
Transaction-based	ATP	ASP*

* Open Transport does not currently provide an implementation of the AppleTalk Session Protocl (ASP).

Addressing

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 using 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. 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" (page 279) in this book. For more information about TCP/IP addresses, see the chapter "TCP/IP Services" (page 237) in this book.

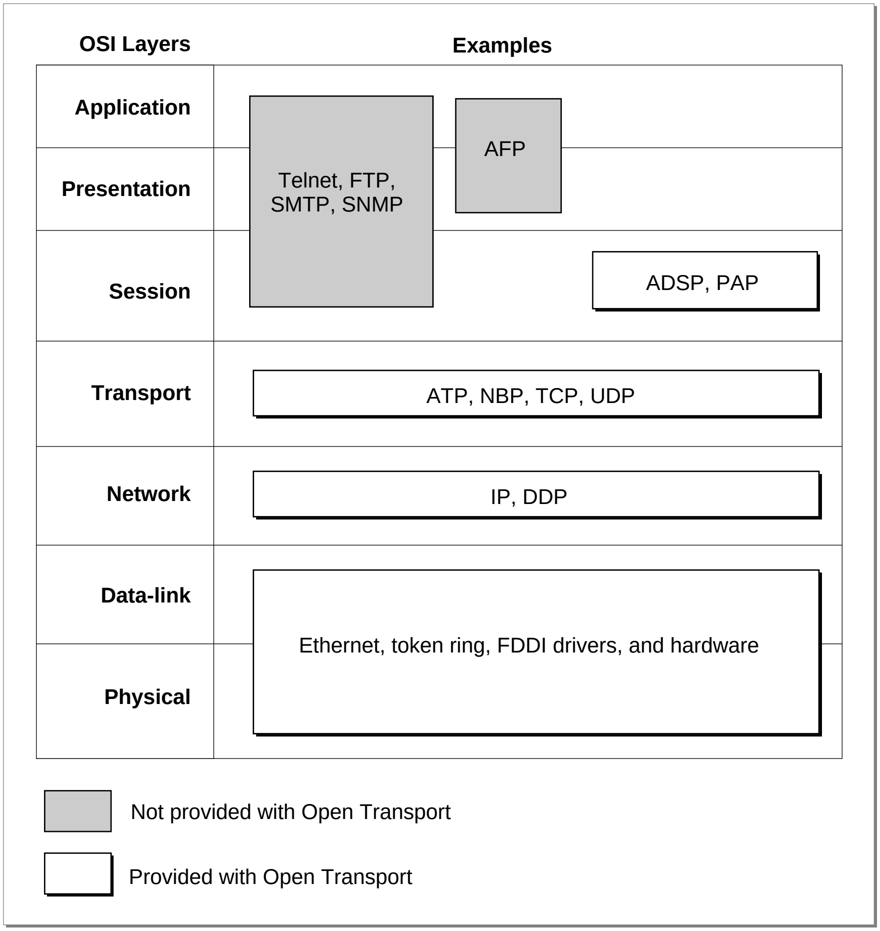
Protocol Stacks and the OSI Model

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.

Figure 1-1 The OSI model and Open Transport protocols



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

- 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 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.

- The **network layer** specifies the network routing of data packets between nodes and the communications between networks, which is referred to as *internetworking.*
- 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 **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 **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 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.

About Networking With Open Transport

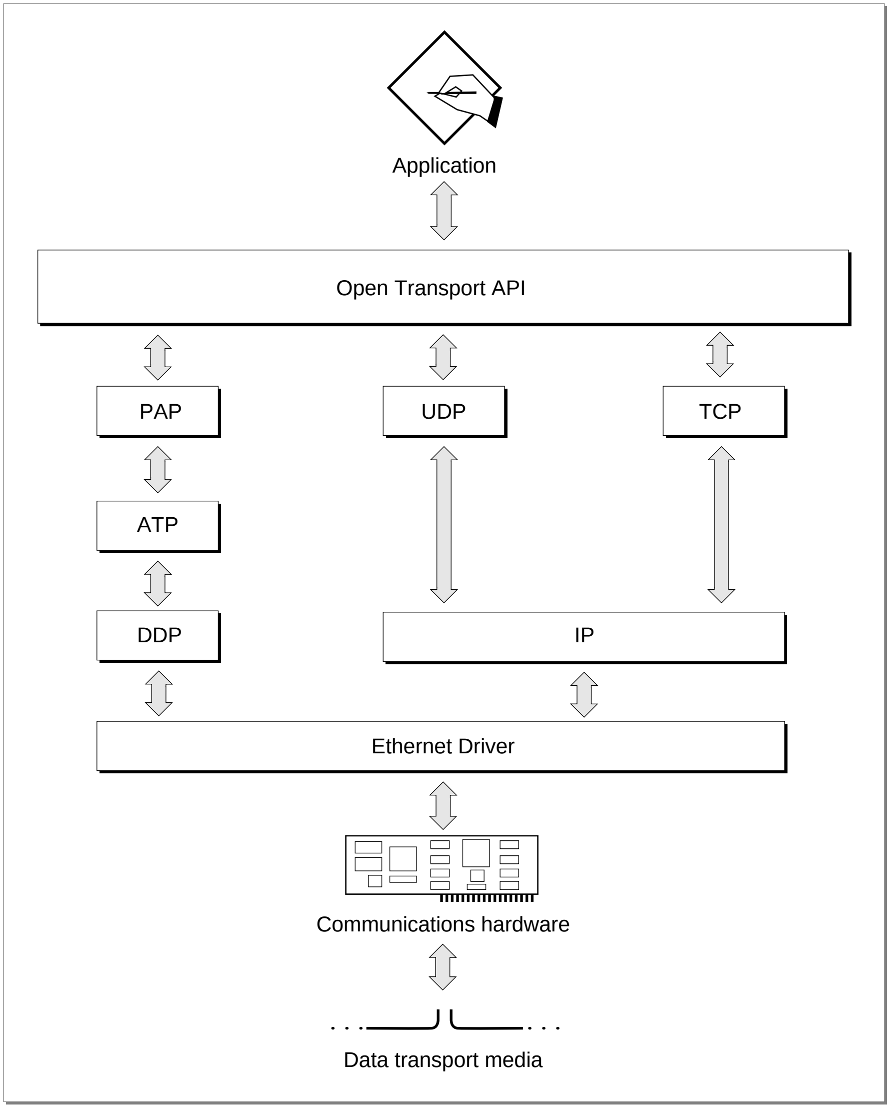
Networking on the Mac OS is implemented through the Open Transport system software. The Open Transport software provides an API that gives you access to the services of the various protocols. The functions you use depend not on the specific protocol you want to use, but on whether the protocol is connection-oriented 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 by Open Transport iself; and **interface functions**, which Open Transport passes through to the underlying software modules. Because the interface functions are implemented by the software modules, the same function might operate somewhat differently depending on the specific modules that execute 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. However, whereas XTI specifies functions only for connectionless and connection-oriented protocols, Open Transport also includes functions for 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 “Option Management” (page 165) 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" (page 61) in this book.

Software Modules

The software modules shown in Figure 1-2 (page 15) 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 receive the reply as a message, all without interfering with any other system activity.

■ 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.

Figure 1-2 (page 15) shows the AppleTalk implementation of the actual STREAMS architecture.

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. This book does not cover STREAMS or writing a STREAMS modules. For more information about STREAMS, see *UNIX System V Release 4: Programmer's Guide: STREAMS* and the *Open Transport Module Devloper's SDK*.

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 Open Transport control panels. Your application can use the default port for a particular protocol or, in some cases, you 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. In the current version, multihoming is supported only with AppleTalk protocols.

Open Transport and Interrupt-Time Processing

Open Transport places severe limitations on functions that can be called at hardware interrupt time and imposes some restrictions on functions that can be called at secondary interrupt time. For a discussion of interrupt-time processing, see “Interrupt-Time Processing” (page 64). For more detailed information, see “Programming With Open Transport” (page 129).

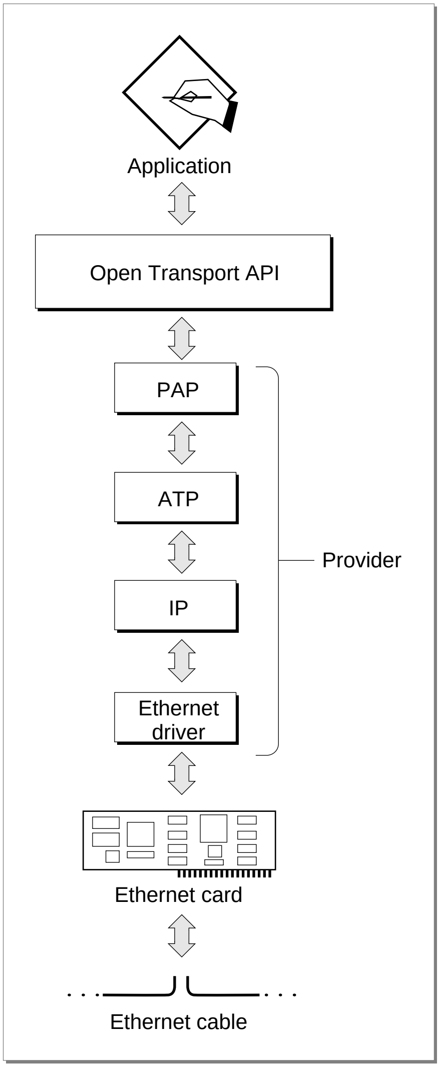
Providers: Endpoints, Mappers, and Services

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 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” (page 22) for more information on the configuration of endpoint providers.

Mapper providers 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. For example, 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 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.

You use **service providers** to handle features unique to a specific type of Open Transport service. For example, 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. Consequently, the TCP/IP service provider functions provide an interface to the DNR.

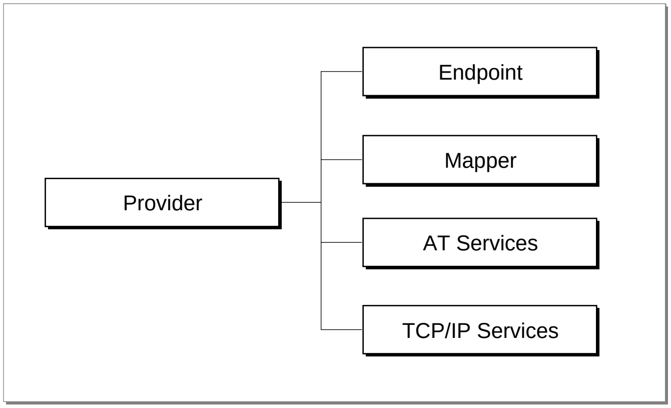
Each provider supports some subset of the standard Open Transport functions, depending on the nature of that provider; for example, an endpoint provider implements different functions than a mapper provider. What's more, a connection-oriented transactionless endpoint provider implements different functions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

Some Open Transport functions are common to all providers. These allow you to open or close a provider, to determine whether a provider executes functions synchronously or asynchronously, to issue a command directly to a STREAMS module underlying a provider, and so on.

When you open an endpoint, mapper, or service provider, 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.

Figure 1-4 shows the hierarchical relationship among Open Transport providers. The C++ API provides classes that mirror this object-oriented hierarchy.

Figure 1-4 Hierarchy of Open Transport providers



Transport Independence

In contrast to earlier Mac OS application programming interfaces (APIs) for AppleTalk and TCP/IP, 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.

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.

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. 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.

Addressing schemes are also protocol-dependent; in order to use specific protocols, you will need to understand these schemes and to use the appropriate protocol-dependent data structure and functions.

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.

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 vendors other than Apple Computer, Inc., you should refer to the documentation that came with that software to determine its use.

There are instances in which the protocol to be used is dictated by the application; for example, HTTP requires TCP. In some cases, you might be in a position to choose the protocol yourself. If so, before you open an endpoint, you should make your choice based on the following issues:

- general purpose or special purpose
- choice of protocol family, AppleTalk or TCP/IP
- connection-oriented or connectionless
- transaction-based or transactionless
- high- or low-level protocol

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 probably need to use the Printer

Access Protocol (PAP). 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 in order 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 a networking technology developed by Apple Computer, Inc. Every Mac OS computer that has ever been made includes AppleTalk hardware and system software. If your application needs to communicate with other Mac OS 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. It offers faster performance compared to AppleTalk and makes cross-platform applications easier to develop. 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 (page 12) shows the protocols provided by Apple Computer, Inc. with Open Transport and where they fit in the OSI model. All the high-level protocols (except UDP) 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 based on AppleTalk or TCP/IP protocols, these are the protocols to use.

The high-level protocol UDP is unusual in that it comines attributes of both high and low level protocols—that is, it does not provide error recovery services but it checks for data corruption.

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 two connection-oriented protocols: ADSP and PAP. ADSP is a full-duplex transactionless protocol, well suited to the transfer of large amounts of data. PAP is a transactionless session-layer protocol and a client of ATP. It is intended primarily for communication with AppleTalk 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 suited to many client-server 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. However, you should keep in mind that transaction-based protocols limit transport independence: currently, only Apple Talk uses these protocols. In addition, given that transaction-based protocols incur some overhead to set up, you might consider choosing one of the connection-oriented protocols instead; these also involve the overhead of establishing the connection but offer more possibility for transport-independence.

Open Transport AppleTalk includes the ATP transaction-based protocols. 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 semi-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 involved in maintaining a session

A workstation application that requires a state-dependent service should use ADSP 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 provides 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 does not include any transaction-based protocols for the TCP/IP protocol family.

Summary

The following is a summary of the preceding sections:

1. If your application requires a specific protocol, use that one.
2. If your intended connectees are local Mac OS computers, use the AppleTalk protocol family.
3. If your intended connectees are not Mac OS computers, or Mac OS computers on a remote network, use the TCP/IP protocol family.
4. To take advantage of Open Transport's transport independence and provide both AppleTalk and TCP/IP, let the user choose.
5. If you need reliability, use a connection-oriented protocol.
6. If you need low overhead or you are writing a real-time application, use a connectionless protocol.
7. Avoid transaction-based protocols.

Getting Started With Open Transport

Contents

This chapter introduces the basic information needed to use Open Transport. If you are writing an application or stand-alone code resource that calls Open Transport functions, you must read the appropriate sections in this chapter to find out how you initialize and close Open Transport, how you configure providers, and how you specify addresses. The code sample shown in Listing 2-4 (page 46) includes material that is described in greater detail in the next two chapters, "Providers" and "Endpoints"; however, it is highly recommended that you read through the sections in this chapter that describe this code. Doing so will enable you to assimilate information in the later chapters more easily and will give you a general sense of what an Open Transport program is like.

The corresponding reference chapter, "Initializing and Closing Open Transport Reference" provides more detailed information about the data structures and functions introduced in this chapter.

Initializing Open Transport

The first step in using Open Transport is to initialize it, and the most practical and efficient way to do that is to call the `InitOpenTransport` function just before you need to call any Open Transport functions. Note that only the client calling Open Transport functions needs to initialize it or close it.

When you initialize Open Transport, it initializes data structures that it needs so that you can call Open Transport functions. An error is returned if Open Transport cannot be used . The following code sample illustrates how you might initialize Open Transport from an application and how you might close it down again upon termination.

```
void main(void)
{
    Boolean gOTInited;
    gOTInited = ( InitOpenTransport() == noErr );

    /* The rest of your application goes here.*/
    if (gOTInited) {
        CloseOpenTransport();
        gOTInited = false;
    }
}
```

Note
If your application needs to manipulate ports or call Open Transport utility functions but it does not need to open or use any providers, you can call the function `InitOpenTransportUtilities` instead of the function `InitOpenTransport`. ◆

Open Transport consists of several parts: the Open Transport kernel, Open Transport utilities, AppleTalk, and TCP/IP. Which of these parts are loaded into memory depends partly on control panel settings and partly on actions you take:

- AppleTalk is loaded if the user has activated it in the control panel. If the user has not activated AppleTalk, it is not possible to load it programmatically.
- TCP/IP is loaded in one of two ways. If the user activates it in the control panel and checks "Load Only When Needed" (the default), TCP/IP is loaded when you open a TCP/IP endpoint or a TCP/IP service provider. If the user activates TCP/IP and checks "Load Only When Needed," TCP/IP is loaded at start-up. If the user does not activate TCP/IP in the control panel, it is not possible to load it programmatically.
- The Open Transport kernel is loaded when AppleTalk or TCP/IP is loaded or when you call the `InitOpenTransport` function.
- The Open Transport utilities are always loaded. You still need to call the function `InitOpenTransportUtilities` to register yourself as an Open Transport client if you want to get or change port information without loading the Open Transport kernel.

Initializing From a Client Application

If you are writing an application, you must follow these steps before you can call any Open Transport functions:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. Call the `InitOpenTransport` function (or the `InitOpenTransportUtilities` function if only accessing port information).

3. Link with the appropriate libraries as described in “Open Transport Libraries” (page 44).

Note
68000 applications do not need to explicitly establish an A5 world before calling Open Transport nor do they need to reset their A5 world before each call to an Open Transport function. This is all done automatically for them. (PowerPC applications never need to be concerned about establishing an A5 world.) ◆

Initializing From a Stand-Alone Code Resource

If you are writing a stand-alone code resource or a shared library, you must follow these steps before calling any Open Transport functions:

1. Include the Open Transport client header file, `OpenTransport.h`.
2. Establish an A5 world if you are writing 68000 code; see the *Apple Shared Library Manager Developer’s Guide* for details on how to do this. (Stand-alone 68000 code resources must ensure that their A5 world is correct each time they call an Open Transport function.)
3. Call the `InitOpenTransport` function (or the `InitOpenTransportUtilities` function).
4. Call the `CloseOpenTransport` function when finished.
5. Link with the appropriate libraries as described in “Open Transport Libraries” (page 44). Remember that a code resource or shared library should link with the “Extn” variants of the libraries.

Using ASLM and Open Transport

Open Transport is based on ASLM and initializes this manager itself. But if your 68000 application uses ASLM, you must

- call the `InitLibraryManager` function before calling the `InitOpenTransport` function
- call the `CloseLibraryManager` function after calling the `CloseOpenTransport` function.

This is true for both applications, shared libraries, and stand-alone resources. For applications the ExitToShell trap will be patched so that the close calls are executed whether you call them explicitly or not. For stand-alone code, you must call the close calls yourself.

Using the Gestalt Function to Determine Whether Open Transport Is Available

If you are writing an installer, you might want to know if Open Transport is available on your computer. To do this, call 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 Gestalt Selector and Response Bits" (page 369).

For version 1.1 or later of Open Transport, you can use the Gestalt function with the 'otvr' selector to determine the Open Transport version in NumVersion format. For more information on Apple's version numbering scheme and the NumVersion format, see *Technote OV12: Version Territory.*

Note
If your application uses Open Transport, it should determine whether it is present using the InitOpenTransport function. Do not use Gestalt for this. The InitOpenTransport functionperforms all the right checks for you. ◆

Configuring and Opening a Provider

After initializing Open Transport but before you can send or receive data, you must configure and open a provider. Providers supply data-oriented services, and are implemented by modules that can be layered to provide the services in which you are interested. To create a provider configuration, you call the function OTCreateConfiguration, passing it a configuration string that describes this layering. The following two sections explain this process in greater detail.

Creating a Configuration Structure

The `OTCreateConfiguration` function creates a configuration structure and returns a pointer to it. The configuration string can be the name of a single protocol, such as “`adsp`”, “`tcp`”, or “`dnr`”, or it can be a comma-separated list of protocol and port names. For instance, the string

```
"adsp,ddp,ltlkB"
```

describes an AppleTalk Data Stream Protocol (ADSP) endpoint provider using the Datagram Delivery Protocol (DDP) with LocalTalk link access provided through 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 (if you have the Printer port configured in the AppleTalk control panel). 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, so Open Transport puts the default Internet Protocol (IP) between TCP and Ethernet.

You can also specify options 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 protocol-specific options and their equivalent string values. But for a simple example, the following configuration string

```
"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. Port names are documented in the chapter “Ports”. 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 AppleTalk control panel.

Most protocols have a literal 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 information on specifying TCP/IP services, see the chapter "TCP/IP Services."

You can use either a constant or a literal value to create a provider that does not use options and that adheres to the default protocol layering. For example, to configure a DDP endpoint, you could use either of the following lines of code:

```
ep = OTOpenEndpoint(OTCreateConfiguration("ddp"), 0, NULL, &err);

ep = OTOpenEndpoint(OTCreateConfiguration(kDDPName), 0, NULL, &err);
```

To configure more complex providers, it is easier to use the literal strings. Using the constant can be confusing, shown by a comparison of the following lines of code:

```
ep = OTOpenEndpoint(OTCreateConfiguration
        ("adsp(EnableEOM=1),ddp,ltlkB"), 0, NULL, &err)

ep = OTOpenEndpoint(OTCreateConfiguration
        (kADSPName"(EnableEOM=1),"kDDPName",ltlkB"), 0, NULL, &err);
```

Some configurations are not valid and the OTCreateConfiguration function will return an error if you try to create one. For example, trying to layer ADSP on top of IP will not work.

Opening a Provider

You can pass the pointer returned by the function OTCreateConfiguration to the function that opens the provider, for example, the OTOpenEndpoint function. Typically, you call the OTCreateConfiguration function inline while calling the function that creates and opens a provider. Here is an example

```
ep = OTOpenEndpoint(OTCreateConfiguration("ddp"), 0, NULL, &err);
```

The function you use to open a provider returns a provider reference. You must specify that provider reference whenever you call a function for that provider. For example, if you open an endpoint provider, you must specify its provider reference when you call a function that sends or receives data.

Reusing Provider Configurations

The functions used to open providers take a pointer to the configuration structure as input, but as part of their processing, they dispose of the original configuration structure. Since typically you use the `OTCreateConfiguration` function to create a single provider at a time, this does not present a problem. Occasionally, however, you may want to reuse a configuration structure to create a second identical provider, or you may want to reuse a configuration 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. Cloning allows you to make multiple copies of the same configuration. For example, you might have only a pointer to a configuration structure, but you want to create ten endpoints, and so you need ten configuration structures. The moment you use the original pointer to create an endpoint, the configuration structure is gone. You can't call the `OTCreateConfiguration` function because you don't have the original configuration string; you were only passed the configuration structure. However, you can clone the original configuration structure before opening each endpoint. For additional information, see "Streamlining Endpoint Creation" (page 144).

Specifying an Address

This section explains the format of Open Transport addresses and the structure used to specify an address. This section also introduces helper routines that can do some of the work in creating these structures for you.

Addressing in Open Transport

Addresses in Open Transport all begin with a common structure that is followed by protocol-specific fields. The common structure is defined by the `OTAddress` type:

```
struct OTAddress
    {
        OTAddressType   fAddressType;
        UInt8           fAddress[1];
    };
typedef struct OTAddress OTaddress;
```

The OTAddress type itself is abstract. You would not declare a structure of this type because it does not contain any address information. However, address formats defined by Open Transport protocols all use the fAddressType field to describe the format of the fields to follow, which do contain address information. For example, the DDPAddress type is an address format used by the AppleTalk protocol:

```
struct DDPAddress
    {
        OTAddressType       fAddressType; /* must be AF_ATALK_DDP */
        UInt16              fNetwork;
        UInt8               fNodeID;
        UInt8               fSocket;
        UInt8               fDDPType;
        UInt8               fpad;
    };
```

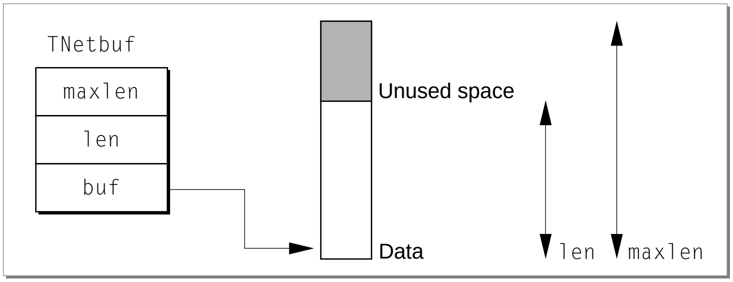
Open Transport recognizes this address as a DDP address because the first field of the address is AF_ATALK_DDP.

Address formats are protocol-specific. The protocol you choose determines the address format that the endpoint you connect to or listen from will accept. For example, if you're using an AppleTalk protocol, you have the choice of using a DDP, an NBP, or a DDP/NBP address format. When you use TCP/IP, you have the choice of the InetAddress format or the DNS address format.

Using TNetBuf Structures

Most provider functions that transfer data pass a parameter of type TNetbuf that specifies the size and location of the data. Such data is usually an address, option information, or the actual data that you want to transfer. You can think of the TNetbuf structure as Open Transport's universal bucket, used to pass and receive 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 amount of data that can be stored in the buffer. 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.

  You may always allocate `TNetbuf` structures for input parameters on the stack.

- 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 making asynchronous calls that use `TNetbuf` structures as output parameters, you should allocate the `TNetBuf` structures (and the buffers they point to) such that they persist until the operation completes. Typically, this means that `TNetbuf` structures for output parameters should only be allocated on the stack if the call is synchronous.

There are two situations in which you would not use a `TNetBuf` structure to store data: when sending noncontiguous data, or when doing a no-copy receive. For additional information about how you should handle these situations, see “Advanced Topics`(page 215)`.”

Storing an Address in a TNetBuf Structure

When you pass an address to Open Transport, you use a TNetBuf structure. Listing 2-1 shows how you might initialize that structure. The listing initializes a DDP address, stores the address in a TNetBuf structure, and then passes that address in the connectCall parameter to the OTConnect function to connect to a remote peer. Note that the addr field is also a TNetBuf structure.

Listing 2-1 Using a TNetBuf structure to store an address

```
void OSStatus MyConnectDDP (EndpointRef ep, UInt16 connectNetworkNumber,
UInt8 connectNodeID, UInt8 connectSockID)
{
    OSStatus err;
    TCall connectCall;
    DDPAddress connectAddr;
    /* initialize the DDP address to connect to */

    connectAddr.fAddressType = AF_ATALK_DDP;
    connectAddr.fNetwork = connectNetworkNumber;
    connectAddr.fNodeID = connectNodeID;
    connectAddr.fSocket = connectSockID;
    connectAddr.fDDPType = 0;

    /* initialize the TNetBuf that contains the address */
    OTMemzero(connectCall, sizeof(TCall));
    connectCall.addr.buf = (UInt8 *) &connectAddr;
    connectCall.addr.len = sizeof(DDPAddress);

    /* now pass the address to Open Transport */
     err = OTConnect(ep, &connectCall, nil);
    return err;
}
```

Using Helper Routines to Initialize an Address

Some Open Transport protocols export routines that make the job of initializing addresses simpler. For example, Listing 2-2 shows how you can use the function OTInitDDPAddress to simplify some the work done in Listing 2-1.

Listing 2-2 Using Helper Routines to Initialize an Address

```
void OSStatus MyConnectDDP (EndpointRef ep, UInt16 connectNetworkNumber,
UInt8 connectNodeID, UInt8 connectSockID)
{
    OSStatus err;
    TCall connectCall;
    DDPAddress connectAddr;

    /* initialize the DDP address to connect to */
    OTInitDDPAddress(&connectAddr, connectNetworkNumber, connectNodeID,
                        connectSockID, 0);

    /* initialize the TNetBuf that contains the address */
    OTMemzero(connectCall, sizeof(TCall));
    connectCall.addr.buf = (UInt8 *) &connectAddr;
    connectCall.addr.len = sizeof(DDPAddress);

    /* now pass the address to Open Transport */
    err = OTConnect(ep, &connectCall, nil);
    return err;
}
```

These helper routines are especially important when an address has a variable length. For example, a DNS address, used by the TCP/IP protocol, is defined as follows:

```
struct DNSAddress
    {
        OTAddressTYpe       fAddressType;   /* always AF_DNS */
        InetDomainName      fName;
    };
typedef struct DNSAddress DNSAddress;
```

The `fName` field of this structure can vary in length. You must pass a DNS address in a `TNetBuf` structure that gives the correct length of the entire address. The helper routine, `OTInitDNSAddress`, not only fills in the fields of the `DNSAddress` structure but also returns the correct length for the `TNetBuf.len` field. This technique is shown in Listing 2-4 (page 46).

Closing Open Transport

This section describes the steps you should take when you no longer need Open Transport. Although the Mac OS provides an automatic clean-up mechanism for applications that call Open Transport functions, it is intended only as a safety net. It's a good idea to do your own clean up, at least for normal application termination. In addition non-application programs are always required to close Open Transport.

System software cannot unload the Open Transport kernel until the last program on the computer that called the `InitOpenTransport` or `InitOpenTransportUtilities` function has also called the `CloseOpenTransport` function. So, if your application only uses the network occasionally, it might be wise to initialize Open Transport only when you need the network, and to close Open Transport immediately after you stop using it.

Closing From Applications

When you are no longer using Open Transport, you can unload the Open Transport software modules by calling the `CloseOpenTransport` function.

It is best if 68000 applications call the `CloseOpenTransport` function, but this will be done automatically if they don't.

Note
If you are running PowerPC applications under version 1.1 (or earlier) of Open Transport, you must call the `CloseOpenTransport` function when terminating. One way to make sure that you do this is to use a CFM terminate procedure in your main application fragment, as shown in Listing 2-3. If you set the appropriate linker option, the system will call the `CFMTerminate` procedure regardless of how your application terminates.

Listing 2-3 CFM terminate procedure

```
static Boolean gOTInited = false;

void CFMTerminate (void);   /* do this if abnormal termination */
{
    if (gOTInited)
    {
        gOTInited = false;
         (void) CloseOpenTransport();
    }
}
void main (void)
{
    OSStatus err;
    err = InitOpenTransport();
    gOTInited = (err ==noErr);

    /* the rest of your application goes here */

    if (gOTInited)              /* do this for normal termination */
        {
         gOTInited = false;
         (void) CloseOpenTransport();
        }
}
```

Note
Open Transport only provides CFM support for 68000 code beginning with version 1.3 ◆

Closing From Stand-Alone Code

For stand-alone code segments, you must call the `CloseOpenTransport` function before you unload from memory. Note that Open Transport only unloads if all clients are done using Open Transport and have called the `CloseOpenTransport` function.

Open Transport Libraries

The libraries that you need to link with vary depending on whether you are writing PowerPC code or 68000 code and on a variety of additional factors, specified in the tables that follow. Table 2-1 lists the libraries you need to link with if you are writing PowerPC code.

Table 2-1 Open Transport libraries for PowerPC code

If you need...	**Link with...**
to build an application	`OpenTransportLib` `OpenTransportAppPPC.o`
AppleTalk services	`OpenTptAppleTalkLib` `OpenTptATalkPPC.o`
Internet services	`OpenTptInternetLib` `OpenTptInetPPC.o`
to use ports or Open Transport Utility functions only	`OpenTransportUtilLib` (instead of `OpenTransportLib`) `OpenTptUtilsAppPPC.o` (instead of `OpenTransportAppPPC.o`)
to build a CFM fragment or ASLM shared library	`OpenTransportExtnPPC.o` (instead of `OpenTransportAppPPC.o`) `OpenTptUtilsExtnPPC.o` (instead of `OpenTptUtilsAppPPC.o`)

Note

If your code is meant to run on machines with and without Open Transport, you should make sure to weak-link with the libraries ending in `Lib`. Otherwise, the system cannot launch your application when Open Transport is not installed. For more information on weak linking, see *Inside Macintosh: PowerPC System Software*. ◆

Table 2-2 lists the libraries you need to link with if you are writing 68000 code. Link with the libraries in square brackets if you are building MPW model-near clients.

Table 2-2 Open Transport libraries for 68000 code

If you need...	Link with...
to build an application	OpenTransport.o [OpenTransport.n.o] OpenTransportApp.o [OpenTransportApp.n.o]
AppleTalk services	OpenTptATalk.o [OpenTptATalk.n.o]
Internet services	OpenTptInet.o [OpenTptInet.n.o]
to use ports or Open Transport utility functions only	OpenTptUtils.o (instead of OpenTransport.o) [OpenTptUtils.n.o] (instead of OpenTransport.n.o)
to build a stand-alone code resource or ASLM shared library	OpenTransportExtn.o (instead of OpenTransportApp.o) [OpenTransportExtn.n.o] instead of OpenTransportApp.n.o.)

Downloading a URL With HTTP

The sample code shown in Listing 2-4 downloads a URL from a web server. It includes two functions: a simple notifier, YieldingNotifier, and a function, MyDownloadHTTPSimple, that downloads the URL. Because the function MyDownloadHTTPSimple contains synchronous calls to Open Transport, the notifier is used to call the function YieldToAnyThread, which cedes time to the processor while a synchronous operation waits to complete. A detailed discussion is contained in the sections following the listing.

The code shown in Listing 2-4 begins by initializing required debugging flags and including the appropriate header files. The OTDebugStr function is not defined in any of the Open Transport header files, but it is exported by the libraries, so its prototype is included.

Listing 2-4 Downloading a URL With HTTP

```
#ifndef qDebug  /* The OT debugging macros in <OTDebug.h> */
#define qDebug 1/* require this variable to be set.*/
#endif

#include <OpenTransport.h>
#include <OpenTptInternet.h> /* header for TCP/IP */
#include <OTDebug.h>    /* header for OTDebugBreak, OTAssert macros */
#include <Threads.h>    /* declaration for YieldToAnyThread */
#include "OTSimpleDownloadHTTP.h" /* header for our own protype */

extern pascal void OTDebugStr(const char* str);

enum
    { kTransferBufferSize = 4096 };/* define size of buffer */

/* define notifier */

static pascal void YieldingNotifier(void* contextPtr, OTEventCode code,
                                       OTResult result, void* cookie)
{
    #pragma unused(contextPtr)
    #pragma unused(result)
    #pragma unused(cookie)
    OSStatus junk;

    switch (code)
    {
        case kOTSyncIdleEvent:
            junk = YieldToAnyThread();
            OTAssert("YieldingNotifier: YieldToAnyThread failed",
                                            junk == noErr);
            break;
        default:
            /* do nothing */
            break;
    }
}


/* Define function that downloads a URL from a web server. */

OSStatus MyDownloadHTTPSimple(const char *hostName,
                            const char *httpCommand,
                            const short destFileRefNum)
{
    OSStatus    err;
    OSStatus    junk;
    Ptr         transferBuffer = nil;
    EndpointRef ep  = kOTInvalidEndpointRef;
    TCall       sndCall;
    DNSAddress  hostDNSAddress;
    OTFlags     junkFlags;
    OTResult    bytesSent;
    OTResult    bytesReceived;
    OTResult    lookResult;
    Boolean     bound       = false;

    /* Allocate a buffer for storing the data as we read it. */

    err = noErr;
    transferBuffer = OTAllocMem(kTransferBufferSize);
    if ( transferBuffer == nil )
         err = kENOMEMErr;

    /* Open a TCP endpoint. */

    if (err == noErr)
    {
        ep = OTOpenEndpoint(OTCreateConfiguration(kTCPName), 0, nil,
                                                            &err);
    }

    /* If the endpoint opens successfully... */

    if (err == noErr)
    {
        junk = OTSetSynchronous(ep);
        OTAssert("MyDownloadHTTPSimple: OTSetSynchronous failed",
                                        junk == noErr);

        junk = OTSetBlocking(ep);

        OTAssert("MyDownloadHTTPSimple: OTSetBlocking failed",
                                        junk == noErr);

        junk = OTInstallNotifier(ep, YieldingNotifier, nil);
        OTAssert("MyDownloadHTTPSimple: OTInstallNotifier failed",
                                        junk == noErr);

        junk = OTUseSyncIdleEvents(ep, true);
        OTAssert("MyDownloadHTTPSimple: OTUseSyncIdleEvents failed",
                                            junk == noErr);

        /* Bind the endpoint. */

        err = OTBind(ep, nil, nil);
        bound = (err == noErr);
    }

    /* Initialise the sndCall structure and call OTConnect. */
    if (err == noErr)
    {
        OTMemzero(&sndCall, sizeof(TCall));
        sndCall.addr.buf = (UInt8 *) &hostDNSAddress;
        sndCall.addr.len = OTInitDNSAddress(&hostDNSAddress, (char *)
                                            hostName);

        err = OTConnect(ep, &sndCall, nil);
    }

    /* Send the HTTP command to the server. */

    if (err == noErr)
        {
            bytesSent = OTSnd(ep, (void *) httpCommand,
                                OTStrLength(httpCommand), 0);
        if (bytesSent > 0)
```

```
            err = noErr;
        else err = bytesSent;
        }
    /* Now we receive the data coming back from the server. */
    if (err == noErr)
    {
        do
        {
            bytesReceived = OTRcv(ep, (void *) transferBuffer,
                                    kTransferBufferSize, &junkFlags);

            if (bytesReceived > 0)
                 err = FSWrite(destFileRefNum, &bytesReceived,
                                                    transferBuffer);
            else err = bytesReceived;

        } while (err == noErr); /* Loop until we get an error. */
    }

    /* Now handle the various forms of error that can occur. */
    if (err == kOTLookErr)
    {
        lookResult = OTLook(ep);

        switch (lookResult)
            {
                case T_DISCONNECT:

                    err = OTRcvDisconnect(ep, nil);
                    break;

                case T_ORDREL:

                    err = OTRcvOrderlyDisconnect(ep);
                    if (err == noErr)
                        {err = OTSndOrderlyDisconnect(ep);}
                    break;

                default:

                    break;
            }
    }

    if ( (err == noErr) && bound )
    {
        junk = OTUnbind(ep);
        OTAssert("MyDownloadHTTPSimple: OTUnbind failed.",
                                            junk == noErr);
    }

    /* Clean up. */
    if (ep != kOTInvalidEndpointRef)
    {
        junk = OTCloseProvider(ep);
        OTAssert("MyDownloadHTTPSimple: OTCloseProvider failed",
                                            junk == noErr);
    }
    if (transferBuffer != nil)
     OTFreeMem(transferBuffer);

    return (err);
}
```

Using Threads for Easy Synchronous Processing

The notifier shown in Listing 2-4 (page 46) is used to yield time to the processor whenever the endpoint receives a `kOTSyncIdleEvent`. Open Transport sends this event whenever it's waiting for a synchronous operation to complete. In response, your notifier should call the function `YieldToAnyThread`.

```
static pascal void YieldingNotifier(void* contextPtr, OTEventCode code,
                                       OTResult result, void* cookie)
{
    #pragma unused(contextPtr)
    #pragma unused(result)
    #pragma unused(cookie)
    OSStatus junk;

    switch (code)
    {
        case kOTSyncIdleEvent:
            junk = YieldToAnyThread();
            OTAssert("YieldingNotifier: YieldToAnyThread failed",
                                        junk == noErr);
            break;
        default:
            /* do nothing */
            break;
    }
}
```

Specifying the Host Names and HTTP Commands

The next section of code in Listing 2-4 (page 46) calls the function MyDownloadHTTPSimple. This function accepts three parameters: the name of the host to which you want to connect, the command to send to the host, and a reference to the file to which you want to download the URL.

```
OSStatus MyDownloadHTTPSimple(const char *hostName,
                              const char *httpCommand,
                              const short destFileRefNum)

/* declarations go here */
The parameter hostName is a pointer to a string that contains the DNS
address of the web server. The DNS address must have the suffix :<port>,
where port is the port number the web server is operating on.
```

The parameter httpCommand is a pointer to the HTTP command to send. Typically this command has the following form, where <*x*> is the URL path.

```
Get <X> HTTP/1.0\0x13\0x10\0x13\0x10
```

For example, if you were to download the URL

```
http://devworld.apple.com/dev/technotes.shtml
```

You would set hostName to devworld.apple.com:80. (The default port for HTTP is 80.) And you would set httpCommand to

```
“GET /dev/technotes.shtml HTTP/1.0\0x13\0x10\0x13\0x10”
```

The parameter destFileRefNum is the reference number of the file to which the results of the HTTP command are written. The function MyDownloadHTTPSimple does not parse the returned HTTP header. The entire incoming stream is written to the file.

Opening an Endpoint and Setting the Mode of Operation

The first section of the function `MyDownloadHTTPSimple` shown in Listing 2-4 (page 46) opens a TCP endpoint, sets the mode of operation, and installs the notifier `YieldingNotifier`.

```
if (err == noErr)
        ep = OTOpenEndpoint(OTCreateConfiguration(kTCPName), 0, nil,
                                                        &err);
        /* If the endpoint opens successfully... */

    if (err == noErr)
    {
        junk = OTSetSynchronous(ep);
        OTAssert("MyDownloadHTTPSimple: OTSetSynchronous failed",
                                                junk == noErr);

        junk = OTSetBlocking(ep);

        OTAssert("MyDownloadHTTPSimple: OTSetBlocking failed",
                                                junk == noErr);

        junk = OTInstallNotifier(ep, YieldingNotifier, nil);
        OTAssert("MyDownloadHTTPSimple: OTInstallNotifier failed",
                                                junk == noErr);

        junk = OTUseSyncIdleEvents(ep, true);
        OTAssert("MyDownloadHTTPSimple: OTUseSyncIdleEvents failed",
                                                    junk == noErr);

        /* Bind the endpoint. */

        err = OTBind(ep, nil, nil);
        bound = (err == noErr);
    }
```

The OTOpenEndpoint function opens the endpoint and returns an endpoint reference (ep). You need to specify this endpoint reference when you set the mode of execution for the endpoint, when you bind the endpoint to an address, and, later, when you establish a connection, receive data, and close the endpoint. The mode of operation for the endpoint is set as synchronous blocking with the call to the OTSetSynchronous function and the OTSetBlocking function. The call to the function OTInstallNotifier installs the notifier YieldingNotifier. The call to the function OTUseSyncIdleEvents tells Open Transport to send kOTSyncIdleEvents to this endpoint; the notifier responds to this event by yielding time to other processes, as noted in “Using Threads for Easy Synchronous Processing” (page 50).

Using a synchronous blocking mode of operation results in a simpler programming model, and the use of the notifier function to yield time to other processes prevents the machine from hanging when synchronous operations are waiting to complete.

Finally, the call to the OTBind function binds the endpoint to a TCP address. (A connection-oriented endpoint can initiate a connection only after the endpoint is bound or queue incoming connection requests.) The second parameter requests the address to which you want to bind the endpoint. In this case, a value of nil is passed; because this is an outgoing connection, it does not particularly matter what address the endpoint is bound to. The third parameter to the OTBind function returns the address to which Open Transport has actually bound the endpoint. The code passes nil because we don’t need that information.

Connecting to the Host and Sending Data

The next section of the function MyDownloadHTTPSimple, shown in Listing 2-4 (page 46), connects to the host and sends an HTTP command to the server. The OTConnect function, which is used to connect the endpoint, passes three parameters: in this case, ep (the endpoint reference), &sndCall (a pointer to the address of the remote endpoint), and a pointer to a buffer in which OTConnect can return information about the connection. Because no data or options were specified with the sndCall parameter, it is not necessary to examine any information returned by the function, so the third parameter is set to nil.

```
if (err == noErr)
{
    OTMemZero(&sndCall, sizeof(TCall));
    sndCall.addr.buf = (UInt8 *) &hostDNSAddress;
```

```
        sndCall.addr.len = OTInitDNSAddress(&hostDNSAddress, (char *)
                                                    hostName);
        err = OTConnect(ep, &sndCall, nil);
    }

    /* Send the HTTP command to the server. */

    if (err == noErr)
    {
         bytesSent = OTSnd(ep, (void *) httpCommand,
                                    OTStrLength(httpCommand), 0);
        if (bytesSent > 0)
            err = noErr;
        else err = bytesSent;
    }
```

Before calling the `OTConnect` function, the `sndCall` structure is initialized. In this case, only the address fields are specified because we are neither sending data with the connection request nor asking for specific option values. The specification of the address is described in detail in "Storing an Address in a TNetBuf Structure" (page 40).

After connecting to the server, the `OTSnd` function is called to send the HTTP command. The `OTSnd` function takes four parameters: in this case, `ep` (the endpoint reference), `httpCommand` (a pointer to the data being sent), `OTStrLength(httpCommand)` (the length of the data), and 0 (specifying that no flags are set). The `OTSnd` function returns the number of bytes sent or a negative number representing an error code if an error occurred. Because the endpoint is in synchronous mode, the function won't return until it has sent all the bytes or it returns an error. The code following the call to `OTSnd` tests to see whether the return value of the function is greater than zero to determine whether or not an error occurred.

Receiving Data From the Remote Endpoint

As shown in Listing 2-4 (page 46) after establishing the connection and sending the data, the function `MyDownloadHTTPSimple` calls the function `OTRcv`, which returns the number of bytes received or a negative (error code) number.

```
if (err == noErr)
    {
        do
        {
            bytesReceived = OTRcv(ep, (void *) transferBuffer,
                                kTransferBufferSize, &junkFlags);

            if (bytesReceived > 0)
                 err = FSWrite(destFileRefNum, &bytesReceived,
                                                transferBuffer);
            else err = bytesReceived;

        } while (err == noErr); /* Loop until we get an error. */
    }
```

Because the endpoint is in synchronous blocking mode, the function won't return until it has received all the data you asked for, or it returns an error. The `OTRcv` function is called repeatedly until it gets an error, which indicates that there is no data left to receive. The function `OTRcv` takes four parameters: in this case `ep` (the endpoint reference), `transferBuffer` (a pointer to the buffer in which data is to be placed), `kTransferBufferSize` (which specifies the size of the buffer), and `&junkFlags` (a pointer to a buffer for flags information), which this sample ignores.

As it receives data, the function `MyDownloadHTTPSimple` calls the `FSWrite` function to write the data to a file.

Error Handling

The next section of the function `MyDownloadHTTPSimple` in Listing 2-4 (page 46) handles errors that might be returned. The most common error is `kOTLookErr`. This indicates that some event has happened that you need to look at. To do this, the function `MyDownloadHTTPSimple` calls the function `OTLook`, which returns an event code for the pending event.

```
if (err == kOTLookErr)
{
        lookResult = OTLook(ep);

        switch (lookResult)
        {
                case T_DISCONNECT:

                    err = OTRcvDisconnect(ep, nil);
                    break;

                case T_ORDREL:

                    err = OTRcvOrderlyDisconnect(ep);
                    if (err == noErr)
                        {err = OTSndOrderlyDisconnect(ep);}
                    break;

                default:
                    break;
        }
}
```

The switch statement includes cases for the most common types of events, T_DISCONNECT and T_ORDREL, and handles them appropriately. The event T_DISCONNECT signals that the remote peer has initiated a disorderly disconnect. HTTP servers will often just disconnect to indicate the end of the data, so all that is needed in response is to clear the event by calling the function OTRcvDisconnect. The event T_ORDREL signals that the remote peer has initiated an orderly disconnect. This means it has no more data to send. In response, your function clears the T_ORDREL event by calling the OTRcvOrderlyDisconnect function and then calls the OTSndOrderlyDisconnect to let the remote peer know that it received and processed the event.

Unbinding the Endpoint and Final Clean-Up

As Listing 2-4 (page 46) shows, having received the data requested, the function MyDownloadHTTPSimple unbinds the endpoint. The conditional call to the OTCloseProvider function closes the endpoint. The following call to the OTFreeMem function frees up memory for the buffer that was allocated to receive data from the OTRcv function.

```
if ( (err == noErr) && bound )
{
    junk = OTUnbind(ep);
    OTAssert("MyDownloadHTTPSimple: OTUnbind failed.",
                                                junk == noErr);
}

/* Clean up. */
if (ep != kOTInvalidEndpointRef)
{
    junk = OTCloseProvider(ep);
    OTAssert("MyDownloadHTTPSimple: OTCloseProvider failed.",
                                        junk == noErr);
}
```

CHAPTER 3

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. This chapter describes 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" (page 5). After reading this chapter, you can read the chapter describing the provider whose services you are interested in. To use the functions described in this chapter, you must first use the `InitOpenTransport` function to initialize Open Transport. This function is described in the chapter "Getting Started With Open Transport" (page 31).

For reference information about the functions and data structures introduced in this chapter, see "Providers Reference" (page 383).

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 example, you can place an encryption module above the AppleTalk Data Stream Protocol (ADSP) module. This combination would provide a stream of network data that was secure from snooping on the network.

Open Transport defines three main types 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. Each protocol family has the option of providing a service provider if one is needed.

In the normal course of events you do not communicate directly with the STREAMS modules that make up a provider. For example, to use 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, or updated.

To use 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 memory. The memory used to create a provider comes partly from your application heap but mostly from the system heap.

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, as with endpoints and mappers, each kind of service (for AppleTalk, TCP/IP, and so on) derives directly from the `TProvider` class; there is no other class for services-type providers. ◆

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 detail in “Providers Reference” (page 383). 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 make a copy of that 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 type of service provider (for AppleTalk, TCP/IP, and so on) has its own type-specific provider functions.

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.

Interrupt-Time Processing

The Open Transport functions that you can call and the means by which you call them vary with the level of execution: system task time, deferred task time, and hardware interrupt time.

In general you can call all Open Transport functions at system task time and most Open Transport functions, asynchronously, at deferred task time. At hardware interrupt time, you are much more limited: you cannot call any of the provider functions and you can call only a small number of Open Transport functions. Software executed at hardware interrupt level includes installable interrupt handlers for NuBus and other devices, Time Manager tasks, VBL tasks, and routines called from within a hardware interrupt handler.

Because it is possible to call many more Open Transport functions from deferred-task level than from hardware-interrupt level, if you need to call an Open Transport function from hardware-interrupt level, you can use the Open Transport function `OTScheduleDeferredTask` or the system function `DTInstall` to have those functions execute at deferred task time. Deferred tasks are scheduled to run when all other hardware interrupt processing is done but before system task processing resumes.

For more information about execution levels and deferred tasks, see *Inside Macintosh: Processes*. For a more detailed view of processing and Open Transport, see Chapter 5, "Programming With Open Transport." For a list of those functions you can call at hardware-interrupt level and deferred-task level, see "Special Functions" (page 793).

Modes of Operation

For each provider, you can use general provider functions to specify how providers execute, whether the provider can block when sending or receving data, and whether endpoint providers acknowledge sends.

A provider can execute in synchronous mode or in asynchronous mode. In **synchronous mode,** provider functions return only when they complete execution. In **asynchronous mode,** they return as soon as they are queued for execution.

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 either waits for as long as it takes to send or receive data (for a synchronous call) or it returns with a result indicating why the operation could not be done immediately (asynchronous call). 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.

A provider's mode of execution and blocking status act together to control the provider's behavior. There are four possible combinations; of these, though only three offer a practical use:

- synchronous blocking

  In this mode, if flow control or other conditions prevent data from being sent or received, the function returns when it is actually able to send or receive the data. Placing a provider in sychronous blocking mode can halt all operations on a Mac OS computer until the operation can complete. For information on how to manage this situation, see "Specifying How Provider Functions Execute" (page 71).

- synchronous nonblocking

In this mode, if flow-control conditions prevent data from being sent or received, the function returns with the result kOTFlowErr or a number indicating only a partial send. Open Transport calls the provider's notifier with a T_GODATA or T_GOEXDATA event when flow control lifts. You must call the function again to continue to send or receive data.

- asynchronous blocking

  In this mode, if flow-control conditions prevent data from being sent or received, the function returns with the result kOTFlowErr or a number indicating only a partial send. Open Transport calls the provider's notifier with a T_GODATA or T_GOEXDATA event when flow control lifts. You must call the function again to send or receive data. If the function cannot complete due to contention for STREAMS resources, it will wait until the required resources are available.

- asynchronous nonblocking

  In this mode, if flow-control conditions prevent data from being sent or received, the function returns with kOTFlowErr, It can also return kEAgainErr. if the function cannot execute as a result of contention for STREAMS resources. Since the point of using asynchronous functions is to be able to continue processing undisturbed until the function returns, using asynchronous nonblocking mode is not practical, as the function might return with the kEAgainErr result a number of times before it actually completes.

A provider's blocking status also governs what happens when you close a provider. In non-blocking mode, closing the provider flushes all outgoing commands in the stream and immediately closes the provider. In blocking mode, the stream is given up to 15 seconds per module to allow outgoing commands to be processed before the stream is closed.

A provider's **send-acknowledgment status** determines whether endpoint functions that send data make an internal copy of the data before sending it. Open Transport ignores the send-acknowledgment status for mapper and service providers.

For specific recommendations about which mode to use and how to set that mode, see "Controlling a Provider's Modes of Operation" (page 70).

Provider Events

Open Transport defines three kinds of events called **provider events.** These events are unique to the Open Transport architecture and are 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 provider event is called an asynchronous event, the second kind is called a completion event, and the third kind is called a miscellaneous 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. 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.

Miscellaneous events are used to notify you or warn you of a change of state in the provider: for example, the provider is about to be closed.

A provider event is identified by a provider event code. These are listed and described in the event codes enumeration (page 383).

- Completion events have a prefix of T_ and the suffix COMPLETE; for example, T_BINDCOMPLETE.
- Asyncronous events have a prefix of T_ and no uniform suffix; for example, T_DATA or T_MEMORYRELEASED.
- Miscellaneous events have a prefix of kOT and no uniform suffix; for example, kOTProviderWillClose.

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 will 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" (page 73) provides additional information about notifier functions and the issues involved in asynchronous processing. You can also refer to "Using Notifier Functions" (page 405) for a description of the notifier functions.

Function Results

Most Open Transport functions return a result of type `OSStatus` or `OTResult`. The main difference between these is that a result of type `OSStatus` is either 0 (`kOTNoError`) or a negative number indicating an error code; a result of type `OTResult` can be either a positive value whose meaning varies with the function called or a negative value indicating an error code. Appendix B (page 785) lists all result codes returned by Open Transport.

- For synchronous function calls, a result of `kOTNoError` indicates that the function succeeded. A negative value indicates an error.
- For asynchronous function calls, if the result code is `kOTNoError`, the operation was successfully started. When the function completes execution, the provider will call the notification function you installed with an event code to indicate which operation completed and a result code indicating whether it succeeded. If an asynchronous function returns any immediate result other than `kOTNoError`, this means that the operation failed before it was started; your notifier will not be called.

The discussion of functions in the reference section of this book describes the meaning of the errors that are most likely to occur for each function. In addition, every Open Transport function might return the result codes listed in Table 3-1. For additional information, please look up the meaning of these result codes in Appendix B (page 785).

Table 3-1 Result codes that all Open Transport functions can return

Result code	Meaning
`kEBADFErr`	The provider reference you supplied is invalid.
`kOTBadSyncErr`	You made a synchronous call at an inappropriate level.
`kENOMEMErr`	There is not enough memory to complete the request.
`kENOSRErr`	There are not enough system resources to complete the request.
`kEAGAINErr` `kEWOULDBLOCKErr`	A provider is in non-blocking mode and Open Transport would have to block to complete the request.
`kOTProtocolErr`	An unspecified protocol error occurred. This is usually fatal. To recover, close the provider.
`kOTClientNotInittedErr`	You have not initialized Open Transport or Open Transport Utilities.
`kOTOutStateErr`	The endpoint is not in an appropriate state for the operation you wish to execute.
`kOTStateChangeErr`	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. 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.

Using Providers

This section explains how you obtain and change a provider’s mode of operation; it provides a 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 that function in Providers Reference(page 383).

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.

Which Mode To Use

Use the following guidelines in determining which mode to use:

- For easiest programming,

  If you are using threads, use synchronous, blocking mode and call the function `OTUseSynchIdleEvents`.

  If you do not use threads, use synchronous, nonblocking and poll for events using the function `GetEndpointState`.

  Using providers in synchronous mode makes for very easy coding; however, if they are also blocking, this could severely affect performance. One way to manage this problem is to call the function `OTUseSyncIdleEvents` (page 410) just after setting the provider's mode of operation. This function generates events of the type `kOTSyncIdleEvents` and sends them to your notifier while Open Transport is waiting to complete a synchronous call. On receipt of this event, your notifier should call the system function `YieldToAnyThread`; this transfers execution to another thread, thus allowing processing to continue while your synchronous operation waits to complete. You should avoid calling functions in synchronous mode at non-System-Task time.

- For best performance, use asynchronous blocking mode.

  Asynchronous processing requires 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" (page 73).

- Never use asynchronous nonblocking mode.

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.

You should be aware that mixing synchronous and asynchronous calls can cause critical problems. Take the following sequence as an example:

1. Set asynchronous mode.
2. Call a function.
3. Set synchronous mode; call a function.
4. The function called in step 2 completes, and the notifier installed for that provider executes at deferred task time.

The problem is that the notifier function, called in step 4, now executes with the provider in synchronous mode: the mode of execution is determined *when* a function is called. Thus any Open Transport function called in the notifier will execute synchronously. However, functions called from a notifier may not execute synchronously; therefore your system will return an error. To avoid this problem, make sure there are no outstanding asynchronous requests when switching to synchronous mode.

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 blocking and you call a function synchronously, all processing on the Macintosh is halted until the synchronous function completes. For information on how to handle this situation, see "Specifying How Provider Functions Execute" (page 71).

If a provider is nonblocking, provider functions that cannot complete send or receive operations return an error indicating the reason. 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 the provider is flow controlled.
- `KENOMEMErr`, indicating that there is not enough memory

In many of 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 the provider not make an internal copy of the data it is sending, but that it relies entirely upon the data being in the buffer you provide. Asking the provider not to make a copy is the same as asking it to acknowledge sends (the Open Transport phrasing). In the current version, you can only specify that *endpoint* providers acknowledge sends. For more detailed information about this mode of operation, see "Acknowledging Sends" (page 215).

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:

- 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. It is typically the `ProviderRef` or a data structure that contains the `ProviderRef`.

- an event code identifying the provider event
- the function result if it's a completion event.
- 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.

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 install the notifier by calling the `OTInstallNotifier` function (page 405). If you want to change notifiers, you must first remove the old notifier by calling the `OTRemoveNotifier` function (page 407) 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. In the notifier shown Listing 3-1 fatal errors all break out of the switch to the default case. The notifier sample is intended to give you a sense of how such code is structured. In general, 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. You should ignore any events you are not expecting.

Listing 3-1 A notifier function

```
static pascal void Notifier(void* context, OTEventCode event, OTResult
result, void* cookie)
{
    EPInfo* epi = (EPInfo*) context;

    switch (event)
    {
        case T_LISTEN:
        {
            DoListenAccept();
            return;
        }

        case T_ACCEPTCOMPLETE:
        {
            if (result != kOTNoError)
                DBAlert1("Notifier: T_ACCEPTCOMPLETE - result %d",
                                                        result);
            return;
        }

        case T_PASSCON:
        {
            if (result != kOTNoError)
            {
                DBAlert1("Notifier: T_PASSCON result %d", result);
                return;
            }
            OTAtomicAdd32(1, &gCntrConnections);
            OTAtomicAdd32(1, &gCntrTotalConnections);
            OTAtomicAdd32(1, &gCntrIntervalConnects);
            if ( OTAtomicSetBit(&epi->stateFlags, kPassconBit) != 0 )
            {
                ReadData(epi);
            }
            return;
        }

        case T_DATA:
        {
            if ( OTAtomicSetBit(&epi->stateFlags, kPassconBit) != 0 )
            {
                ReadData(epi);
            }
            return;
        }

        case T_GODATA:
        {
            SendData(epi);
            return;
        }

        case T_DISCONNECT:
        {
            DoRcvDisconnect(epi);
            return;
        }

        case T_DISCONNECTCOMPLETE:
        {
            if (result != kOTNoError)
                DBAlert1("Notifier: T_DISCONNECT_COMPLETE result %d",
                                result);
            return;
        }

        case T_MEMORYRELEASED:
        {
```

```
                OTAtomicAdd32(-1, &epi->outstandingSends);
                return;
            }

            default:
            {
                DBAlert1("Notifier: unknown event <%x>", event);
                return;
            }
        }
    }
```

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.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.

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

- might need to be reentrant
- cannot move or purge memory
- cannot depend on the validity of handles to unlocked blocks
- should not perform time-consuming tasks
- should not make synchronous calls to Open Transport
- should not make synchronous Device Manager or File Manager calls

The only exception to these rules occurs when you are responding to the event `kOTProviderWillClose`. See the event codes enumeration (page 383) for additional information.

Open Transport might call a notification routine reentrantly. Open Transport attempts to queue calls to a notification routine to prevent reentrancy and to keep the processor stack from growing, but this behavior is not guaranteed. You should be prepared and write your notification routine defensively. For additional information, see “MyNotifierCallbackFunction” (page 413).

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 return parameters is not overwritten. For example, you could make these variables global or use the function `OTAllocMem` to allocate them.

Transferring a Provider's Ownership

An Open Transport client is any task that calls the `InitOpenTransport` function. 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 and code resources, 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 390). 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 invalid and should not be used. Listing 3-2 furnishes an example of transferring a provider's ownership. In this example, an Open Transport client, the ProviderFactory library, creates an endpoint. It then passes the endpoint reference back to another Open Transport client, the TransferProvider application. The application is responsible for transferring the ownership of the endpoint from the library to itself before shutting down the library; it does so using the `GetProviderFromFactory` function.

Listing 3-2 Transferring provider ownsership

```
static OSStatus GetProviderFromFactory(void)
{
    OSStatus err;
    EndpointRef originalEndpoint;
    OTClient originalOwner;
    EndpointRef newEndpoint;
    TEndpointInfo newEndpointInfo;

/* Use the factory library to create an endpoint.*/

    err = FactoryCreateEndpoint(&originalEndpoint, &originalOwner);

    if (err == noErr) {

/* Transfer the ownership of endpoint, so that OT knows we now own it */

        newEndpoint = OTTransferProviderOwnership(originalEndpoint,
                                originalOwner, &err);

        if (err == noErr) {

/* We can now use newEndpoint as if we created it.  */
/* We call OTGetEndpointInfo as an example of an operation */
/*  on the endpoint.*/

            err = OTGetEndpointInfo(newEndpoint, &newEndpointInfo);

            if (err == noErr) {
                printf("Maximum size of endpoint address = %ld.\n",
                                        newEndpointInfo.addr);
            }

            OTCloseProvider(newEndpoint);
        }
    }

    return err;
}

void main(void)
{/* initialize connection to Open Transport */
    OSStatus err;

    err = InitOpenTransport();
    if (err == noErr) {

/* initialize the provider factory library. */
        err = InitProviderFactory();

        if (err == noErr) {
        /* call GetProviderFromFactory to demonstrate */
        /* use of OTTransferProviderOwnership */
            err = GetProviderFromFactory();

            CloseProviderFactory();
        }

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

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 or a `kOTProviderIsClosed` event.

If you get a kOTProviderIsClosed event, the service underlying your provider is already gone; closing the provider only frees up memory resources.

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 you have opened a provider asynchronously (for example, by calling the AsyncOpenEndpoint function), it is not possible to close it before the call has completed. This might happen if the user quits the application before the provider has opened. For this reason, it is safer to open a provider synchronously and then to use the OTSetAsynchronous function to set the execution mode.

The blocking status of a provider governs what happens when the provider is closed. In non-blocking mode, closing the provider flushes all outgoing commands in the stream and immediately closes the provider. In blocking mode, the stream is given up to 15 seconds per module to allow outgoing commands and data to be processed before the stream is closed.

If you are closing a provider in response to a kOTProviderWillClose event, note that Open Transport issues this event only at system task time. Thus, 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.

CHAPTER 4

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
- 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” (page 5) and “Providers” (page 61), which introduce many of the concepts discussed and further elaborated in this chapter.

This chapter offers minimal information about options, values you can specify to control the behavior of providers; these values are set and sometimes retrieved by clients. For information about options, you must read the chapter “Option Management” (page 165).

After you are familiar with the concepts described in this chapter, you can get additional information about transferring data and improving performance in “Programming With Open Transport” (page 129) and in “Advanced Topics” (page 215).

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 section “Configuring and Opening a Provider” (page 34) 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,** a 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 protocol 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 depend 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. 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 Type of Service

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

- connection-oriented transactionless service

  This type of service allows for the transfer of very large amounts of data with guaranteed data delivery and a reliable data stream. ADSP and TCP provides this type of service.

- connection-oriented transaction-based service

  This type of service allows you to conclude an unlimited number of parallel transactions. It guarantees delivery and can detect duplicate sends. ASP (not currently implemented) provides this type of service.

- connectionless transactionless service

  This type of service provides best-effort delivery and allows the transfer of relatively small amounts of data at one time.Some protcols can calculate checksums for incoming packets; IP and DDP provide this type of service.

- connectionless transaction-based service

  This type of service allows for the transfer of larger amounts of data than connectionless transactionless service. It also provides some error checking, detects duplicate sends, and guarantees that response packets are delivered in the order sent. ATP provides this type of service.

The chapter “Introduction to Open Transport” (page 5) defines and describes the different services that each protocol 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 type of service is implemented for your endpoint and the options that you can use to fine-tune its behavior. The section “Using Endpoints” (page 99) 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 type 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, or modes of execution with all endpoint types. However, the type of the endpoint determines which endpoint functions you can call to transfer data. There is no single function that you can use to send data or to receive data. 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 4-1 presents a summary of the function names for functions used to transfer data. The functions are grouped together based on the endpoint's type of service.

Table 4-1 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 the5 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 `OTSndURequest` 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** (or "orderly release") 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 abstract basic types of service offered by network protocols. For example, ADSP and TCP both offer connection-oriented transactionless service. As a result, changing the endpoint to a different protocol (but one that provides the same type of service) would require 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 you can set for an endpoint, to link the behavior of the provider more closely to a specific protocol. By using options, you can take advantage of a feature that is unique to a protocol.

In general, the use of options decreases the transport independence 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. 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.** 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 "Option Management" (page 165).

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

The chapter "Providers" (page 61) explains these concepts and describes the functions you use to get and set a provider's mode of operation.

One thing to keep in mind is that not all endpoint functions operate differently in asynchronous mode. Those functions that do behave differently are listed in Table 4-2. For each function, the table lists the corresponding completion event.

Table 4-2 Endpoint functions that behave differently in synchronous and asynchronous modes

Function	Completion event
`OTOptionManagement`	`T_OPTIONMANGEMENTCOMPLETE`
`OTBind`	`T_BINDCOMPLETE`
`OTUnbind`	`T_UNBINDCOMPLETE`
`OTAccept`	`T_ACCEPTCOMPLETE`

Table 4-2 Endpoint functions that behave differently in synchronous and asynchronous modes (continued)

Function	Completion event
OTSndRequest	T_REQUESTCOMPLETE
OTSndReply	T_REPLYCOMPLETE
OTSndURequest	T_REQUESTCOMPLETE
OTSndUReply	T_REPLYCOMPLETE
OTSndDisconnect	T_DISCONNECTCOMPLETE
OTGetProtAddress	T_GETPROTADDRCOMPLETE
OTResolveAddress	T_RESOLVEADDRCOMPLETE

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.

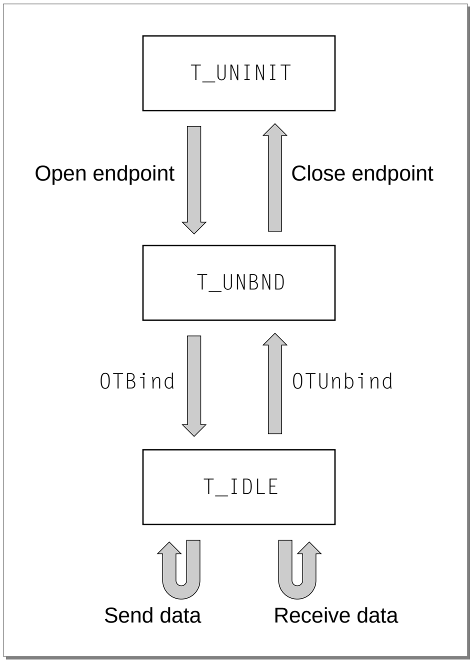
The endpoint's type of service determines the possible states an endpoint can be in while it is transferring data. For example, a connectionless endpoint can only transfer data while it is in the T_IDLE state, and a connection-oriented endpoint can only transfer data while it is in the T_DATAXFER state. Table 4-3 describes possible endpoint states for connectionless and connection-oriented endpoints and suggests in parentheses an English equivalent for the name of each constant.

Table 4-3 Endpoint states

State	Meaning
T_UNINIT	Uninitialized. This endpoint has been closed and destroyed or has not been created.
T_UNBND	Unbound. This endpoint is initialized but has not yet been bound to an address.
T_IDLE	Idle. This endpoint has been bound to a local protocol 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	Outgoing connection request. This connection-oriented endpoint has initiated a connection and is waiting for the remote peer to accept the connection.
T_INCON	Incoming connection request. This connection-oriented endpoint has received a connection request but has not yet accepted or rejected the request.
T_DATAXFER	Data transfer. This connection-oriented endpoint can now transfer data because the connection has been established.
T_OUTREL	Outgoing release request. This connection-oriented endpoint has issued an orderly disconnect that the remote peer has not acknowledged. The endpoint can continue to read data but must not send any more data.
T_INREL	Incoming release request. 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.

Figure 4-1 shows a diagram illustrating the possible endpoint states for a connectionless endpoint.

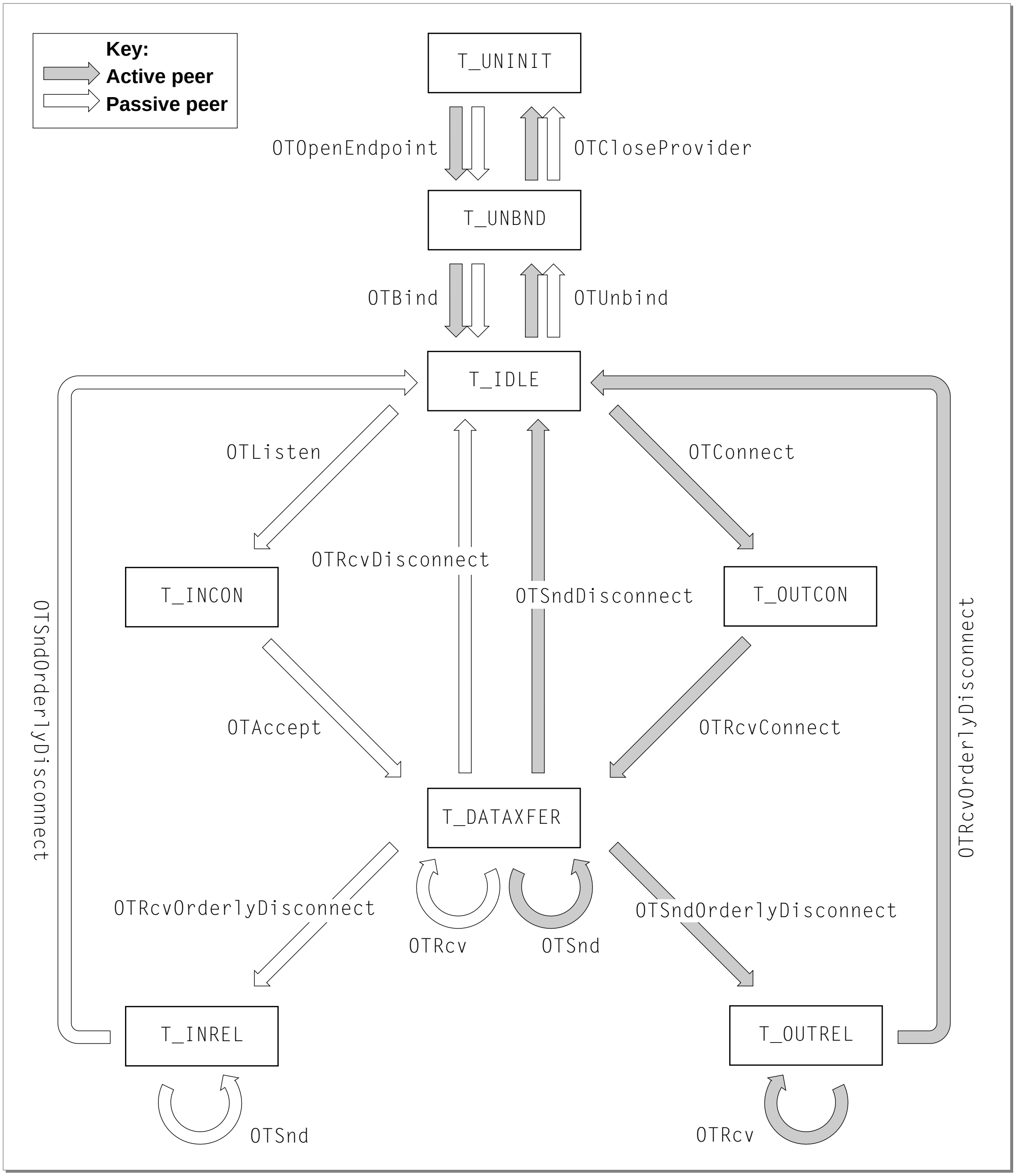
Figure 4-1 Typical endpoint states for a connectionless endpoint



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 4-2 shows a state diagram illustrating the possible endpoint states for a connection-oriented endpoint.

Figure 4-2 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 (not shown in Figure 4-2). 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 OTRcvDisconnect function rejects the connection and places the active peer in the T_IDLE state (not shown in Figure 4-2). 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 you have finished transferring data, you should tear down the connection by using an orderly disconnect process if possible. That is, you should check to see whether the protocol supports an orderly disconnect. If it does, you initiate this process by calling the OTSndOrderlyDisconnect function. This places the calling endpoint in the T_OUTREL state. It also creates a pending T_ORDREL event for the other endpoint. The endpoint to which you have sent the disconnection request can become aware of the event by means of a notifier function or by calling 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 you must also acknowledge. Disconnecting the endpoints places them in the T_IDLE state again, and you can reconnect or close them.

Note
It is not required that the active peer in the disconnect phase be the same as the active peer in the connect pahse ◆.

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.

Table 4-4 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 4-4 Functions that can change an endpoint's state

		State after call	
Function	**Valid state before calls**	if call succeeds	if call fails
`OTOpenEndpoint`	`T_UNINIT`	`T_UNBND`	n/a
`OTAsyncOpenEndpoint`	`T_UNINIT`	`T_UNBND`	n/a
`OTBind`	`T_UNBND`	`T_IDLE`	`T_UNBND`
`OTUnbind`	`T_IDLE`	`T_UNBND`	
`OTGetEndpointInfo`	`any but T_UNINIT`		
`OTGetEndpointState`	`any but T_UNINIT`		
`OTLook`	`any but T_UNINIT`		
`OTGetProtAddress`	`any but T_UNINIT`		
`OTResolveAddress`	`any but T_UNINIT`		
`OTSync`	`any but T_UNINIT`		
`OTAlloc`	`any but T_UNINIT`		
`OTFree`	`any but T_UNINIT`		
`OTCountDataBytes`	`T_IDLE`		
`OTSndUData`	`T_IDLE`		
`OTRcvUDErr`	`T_IDLE`		

Table 4-4 Functions that can change an endpoint's state (continued)

		State after call	
Function	**Valid state before calls**	if call succeeds	if call fails
OTRcvUData	T_IDLE		
OTSndURequest	T_IDLE		
OTSndUReply	T_IDLE		
OTRcvUReply	T_IDLE		
OTCancelURequest	T_IDLE		
OTCancelUReply	T_IDLE		
OTConnect	T_IDLE	T_OUTCON	T_IDLE
OTListen	T_IDLE, T_INCON		
OTAccept			
--destination	T_IDLE, T_UNBND	T_DATAXFER	T_IDLE, T_UNBND
--source	T_INCON	T_INCON T_IDLE	T_INCON
OTSnd	T_DATAXFER, T_INREL		
OTRcv	T_DATAXFER, T_OUTREL		
OTSndRequest	T_DATAXFER, T_INREL		
OTRcvRequest	T_DATAXFER, T_OUTREL		
OTSndReply	T_DATAXFER, T_OUTREL		
OTRcvREply	T_DATAXFER, T_INREL		
OTCancelRequest	T_DATAXFER		
OTCancelReply	T_DATAXFER		
OTSndDisconnect	T_DATAXFER, T_INREL, T_OUTCON, T_OUTREL	T_IDLE	state before call
	T_INCON	T_IDLE, T_INCON	T_INCON

Table 4-4 Functions that can change an endpoint's state (continued)

		State after call	
Function	**Valid state before calls**	if call succeeds	if call fails
OTRcvDisconnect	T_DATAXFER, T_INREL, T_OUTCON, T_OUTREL	T_IDLE	state before call
	T_INCON	T_IDLE, T_INCON	T_INCON
OTSndOrderlyDisconnect	T_DATAXFER	T_OUTREL	T_DATAXFER
	T_INREL	T_IDLE	T_INCON
OTRcvOrderlyDisconnect	T_DATAXFER	T_INREL	T_DATAXFER
	T_OUTREL	T_IDLE	T_OUTREL

The arrival of an asynchronous event can also change the state of an endpoint. Table 4-5 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 4-5 Events that can change an endpoint's state

Old state	**Event**	**New state**	**Consuming function**
T_IDLE	T_LISTEN	T_INCON	OTListen
T_IDLE	T_CONNECT	T_DATAXFER	OTRcvConnect
T_IDLE T_UNBND	T_PASSCON	T_DATAXFER	none
T_OUTCON T_DATAXFER T_OUTREL T_INREL	T_DISCONNECT	T_IDLE	OTRcvDisconnect
T_DATAXFER	T_ORDREL	T_INREL	OTRcvOrderlyDisconnect
T_INCON	T_DISCONNECT	T_IDLE T_INCON	OTRcvDisconnect

The section “Handling Events for Endpoints” (page 102) 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 data 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**) 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. 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.

Three special constants are used to specify information about TSDUs and ETSDUs:

- T_INFINITE specifies that there is no limit to the size of a TSDU
- T_INVALID means that the endpoint does not support this type of data
- 0 means that TSDUs are not supported

For additional information, see “The TEndpointInfo Structure” (page 426).

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 E ndpoints” (page 101) 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 type of service.

No matter what type of service you want to implement, you must read the sections "Opening and Binding Endpoints," "Obtaining Information About E ndpoints," "Handling Events for Endpoints," and "Sending and Receiving Data." After you have read these sections, you can read the section describing the type of service you are interested in implementing.

Table 4-6 shows how some of the Open Transport protocols fit with an endpoint's type of service. For example, if you want to use ATP, you would need to read the section "Using Connectionless Transaction-Based Service" (page 123). If you want to use ADSP, you would need to read the section "Establishing and Terminating Connections" (page 105) and the section "Using Connection-Oriented Transactionless Service" (page 120).

Table 4-6 The Open Transport type-of-service matrix and some Open Transport protocols

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

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 classic 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

After you 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` functions, which are described in "Providers" (page 61).

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. The chapter "Getting Started With Open Transport" (page 31), contains information about creating a configuration structure for an endpoint provider.

If you use the `OTAsyncOpenEndpoint` function to open an endpoint, you also specify 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 (page 405) to install your notifier function.

When Open Transport creates an endpoint, it establishes important state information for the endpoint, including information about

- the endpoint's modes and type of service

- 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

You can retrieve this information by calling functions that return information about the endpoint. These functions are described in the next section, "Obtaining Information About E ndpoints" (page 101).

When the function you use to open the endpoint returns, it passes back to you an endpoint reference. You can 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" (page 111).

No matter what type 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 E ndpoints" (page 101), and "Handling Events for Endpoints" (page 102) After you read these sections, you can proceed by reading about the type of service you want to implement.

Obtaining Information About E ndpoints

While you can use general provider functions to determine an endpoint's mode of execution, you must use endpoint-specific functions to obtain the endpoint's type of service, state, or address.

The TEndpointInfo structure contains most of the information you need to determine how you can use an endpoint. You can obtain a copy of this structure when you open the endpoint, or by calling the OTGetEndpointInfo function. 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 type 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.

To obtain information about an endpoint's state, you call the function OTGetEndpointState. This function returns a positive integer indicating the endpoint state or a negative integer corresponding to a result code. Table 4-3 (page 90) lists and describes endpoint states. If the endpoint is in asynchronous mode and you are not using a notifier function, you can 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 4-4 (page 95).

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 corresponding to a higher-level address for the endpoint's protocol. 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. This manual explains these formats for the TCP/IP and the AppleTalk protocols. For information about obtaining the addresses that correspond to a name pattern, see "Mappers" (page 149).

Handling Events for Endpoints

Like other providers, endpoint providers can operate synchronously or asynchronously. If you use an asynchronous endpoint, you need to create a notifier function that the provider can call to inform you of provider events. This section describes how you handle events for endpoint providers.

Event handling for endpoints is basically the same as that described for providers in "Provider Events" (page 67). 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.

IMPORTANT

An endpoint does not receive any more T_DATA events until its current T_DATA event is cleared. ▲

Table 4-7 lists the functions you use to clear pending asynchronous events.

Table 4-7 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, OTLook
T_GOEXDATA	OTSnd, OTLook
T_PASSCON	none

This works as follows. A 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, the provider notifies the the client. If the queue is not empty, then no notification is delivered 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 examines 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, the provider delivers a new event to the client. Typically, this new event is delivered to the client just before the function returns which removed the head element of the queue.

Clearing Events and Synchronization Problems

You do not have to issue calls that clear events 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 is not guaranteed to be issued.

One exception to this rule occurs when dealing with transaction-based protocols. When the client gets a `T_REPLY` event, it should call the function `OTRcvUReply` until the function returns the result `kOTNoDataErr`. If the client calls `OTRcvUReply` from the foreground (rather than from a notification routine), 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.

Notifier Reentrancy

Open Transport endpoints are handled independently. That means that you can use the same code for the notifiers of two different endpoints. A different instance of the notfier is invoked for each endpoint using the notifier.

Whatever events are pending on one endpoint have (for the most part) no effect on any other endpoints. For example, assume that an endpoint is notified of a `T_DATA` event. Following this, a separate `T_DATA` event is sent to a second endpoint. The notifier for the second endpoint is invoked, interrupting the first endpoint's processing of its `T_DATA` event. For additional information, see "Using Asynchronous Processing With a Notifier" (page 134).

Polling for Events

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 4-8 lists the functions that can return the result kOTLookErr when the corresponding event is pending.

Table 4-8 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_GODATA, 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 4-7 (page 103) to clear the event, and then you can retry the function that failed.

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, but the calls you use to transfer data differ. The section "Using Connection-Oriented Transactionless Service" (page 120) explains how you transfer data once you have established a connection.

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
The value you specify for the `qlen` field indicates your desired value, but Open Transport may 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. After calling the `OTBind` function, you might want to take a look at this field to see whether the protocol has imposed a limit on the `qlen` value. ◆

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" (page 111).

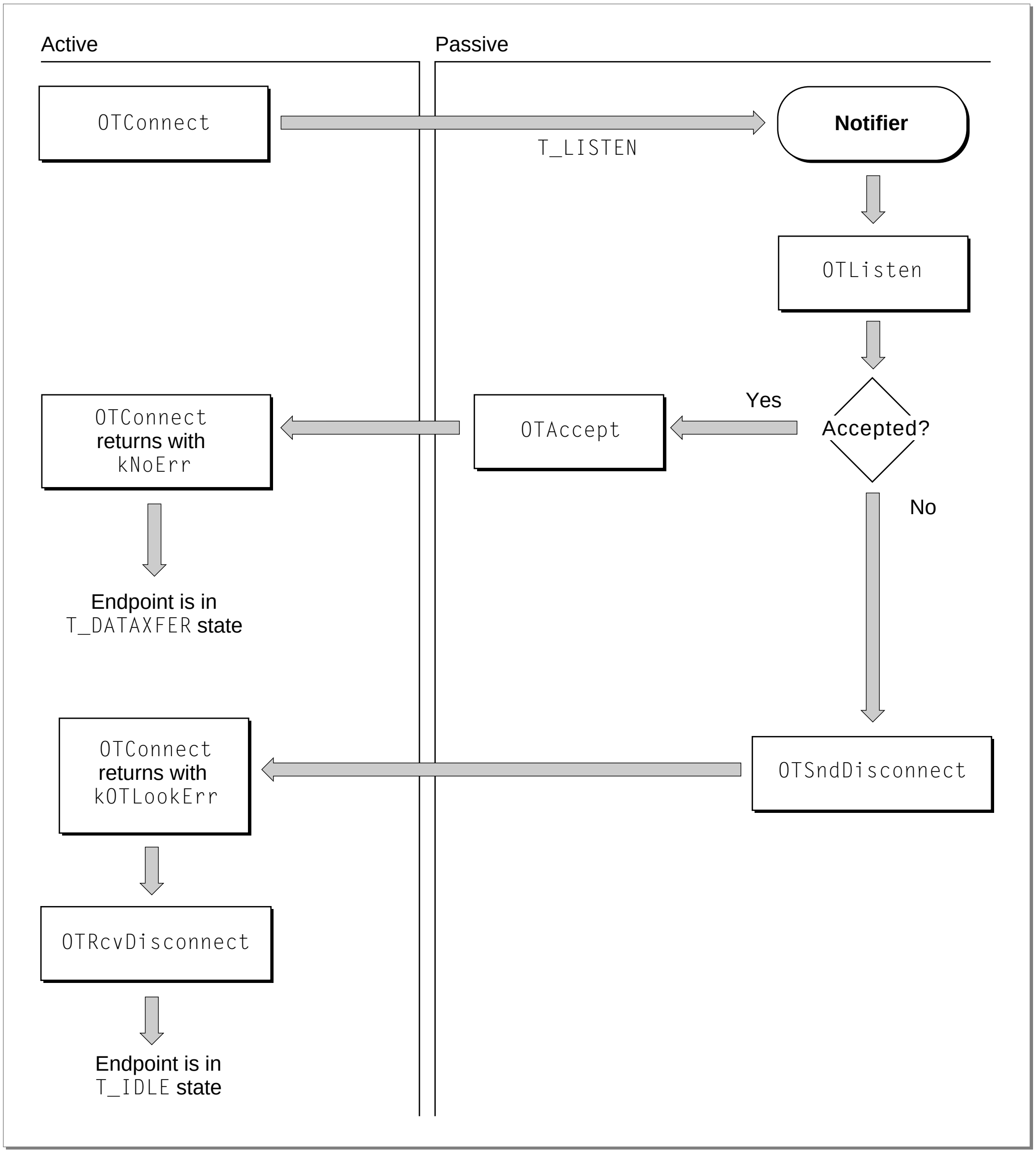
Establishing a Connection

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`	Gets information about 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.
`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 4-3 illustrates the process of establishing a connection in synchronous mode.

Figure 4-3 Establishing a connection with the active peer in synchronous mode

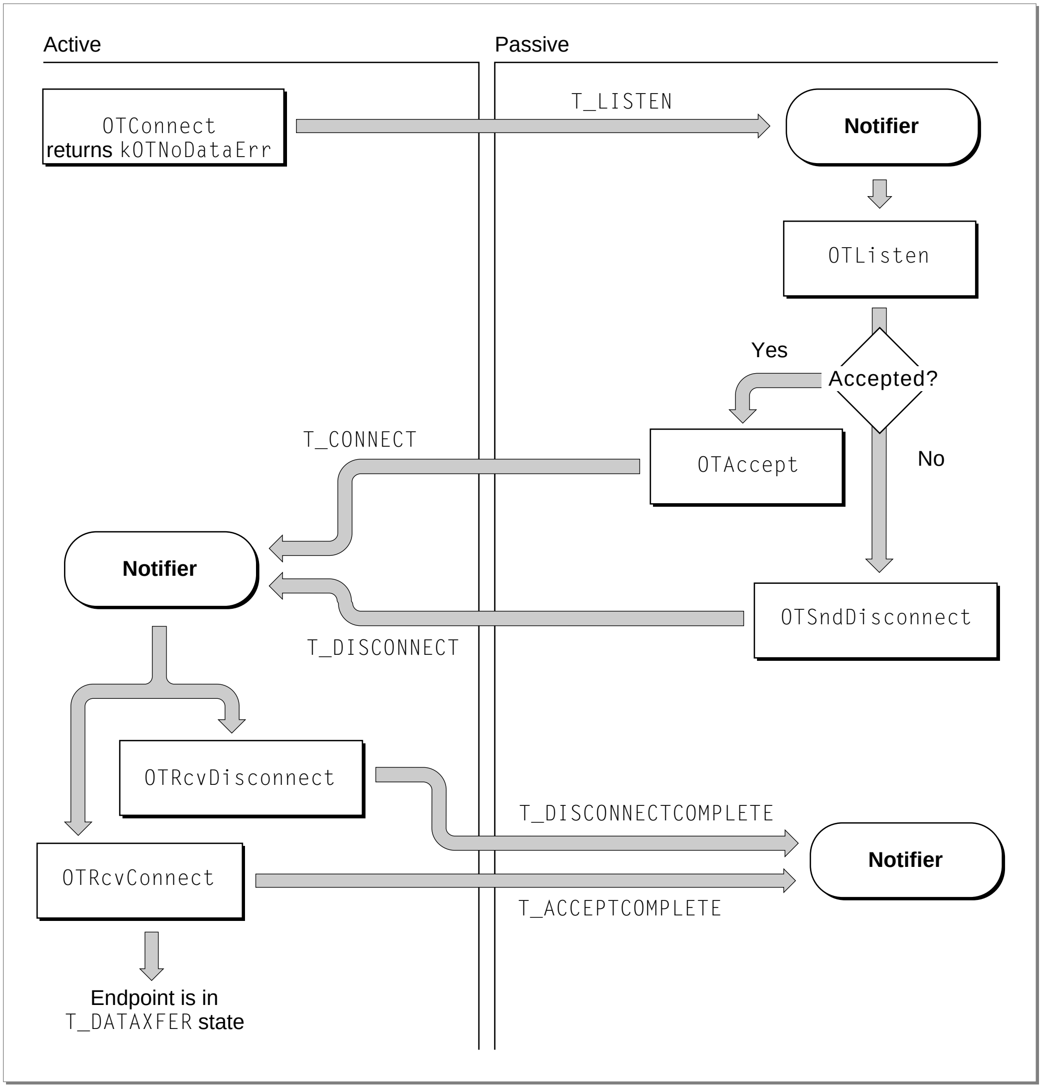

Active
Passive
OTConnect
T_LISTEN
Notifier
OTListen
Yes
Accepted?
No
OTConnect returns with kNoErr
OTAccept
Endpoint is in T_DATAXFER state
OTConnect returns with kOTLookErr
OTSndDisconnect
OTRcvDisconnect
Endpoint is in T_IDLE state

As Figure 4-3 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 kOTNoErr. If the passive peer rejects the connection or the request times out, the OTConnect function returns with kOTLookErr. When the OTConnect function returns, the active peer must examine the result. If the call succeeded, it should begin to transfer data. If the call failed, the active peer should call the OTRcvDisconnect function to restore the endpoint to a valid state for subsequent operations.

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

Figure 4-4 Establishing a connection in asynchronous mode



The active peer calls the OTConnect function, which returns right away with a code of kOTNoDataErr. 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 rejected 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. 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 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.
- The underlying implemention determines whether you must bind the endpoint accepting a connection to the address of the endpoint listening for the connection. In general, it is recommend that you do not bind it to the same address.

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.

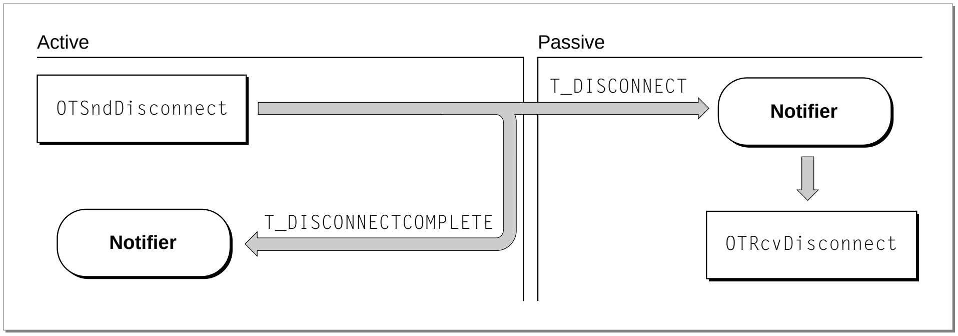
Terminating a Connection

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. For additional information about handling an unexpected disconnection from a remote client, see “Handling Dead Clients” (page 145).

Using an Abortive Disconnect

You use the OTSndDisconnect and OTRcvDisconnect functions to perform an abortive disconnect. Figure 4-5 illustrates the process for two asynchronous endpoints. The figure shows the active peer initiating the disconnection; in fact, either peer can initiate the disconnection.

Figure 4-5 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 know 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.

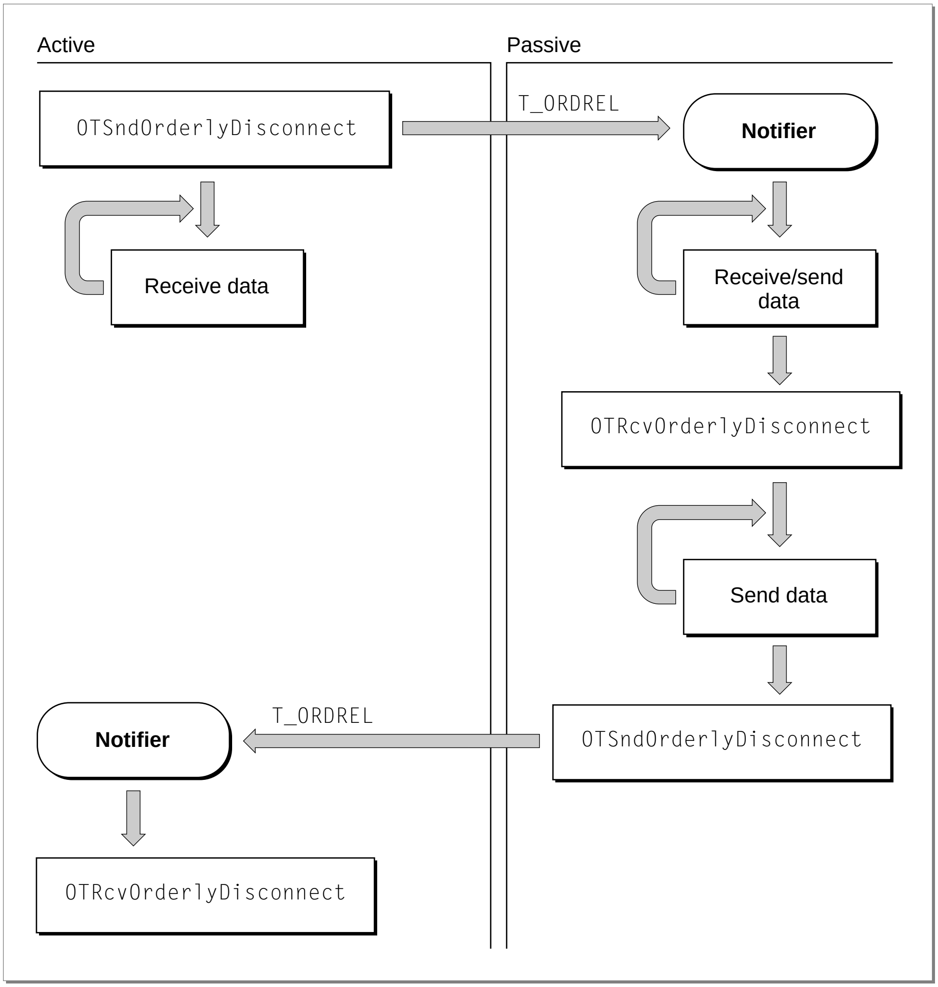
When the connection has been closed, the endpoint provider calls the active peer's notifier, passing T_DISCONNECTCOMPLETE for the event parameter. At this time the endpoint is once more in the T_IDLE state.

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 disconnect. It is safest to assume, unless you know for certain it to be otherwise, that the endpoint supports only local orderly disconnects.

Figure 4-6 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.

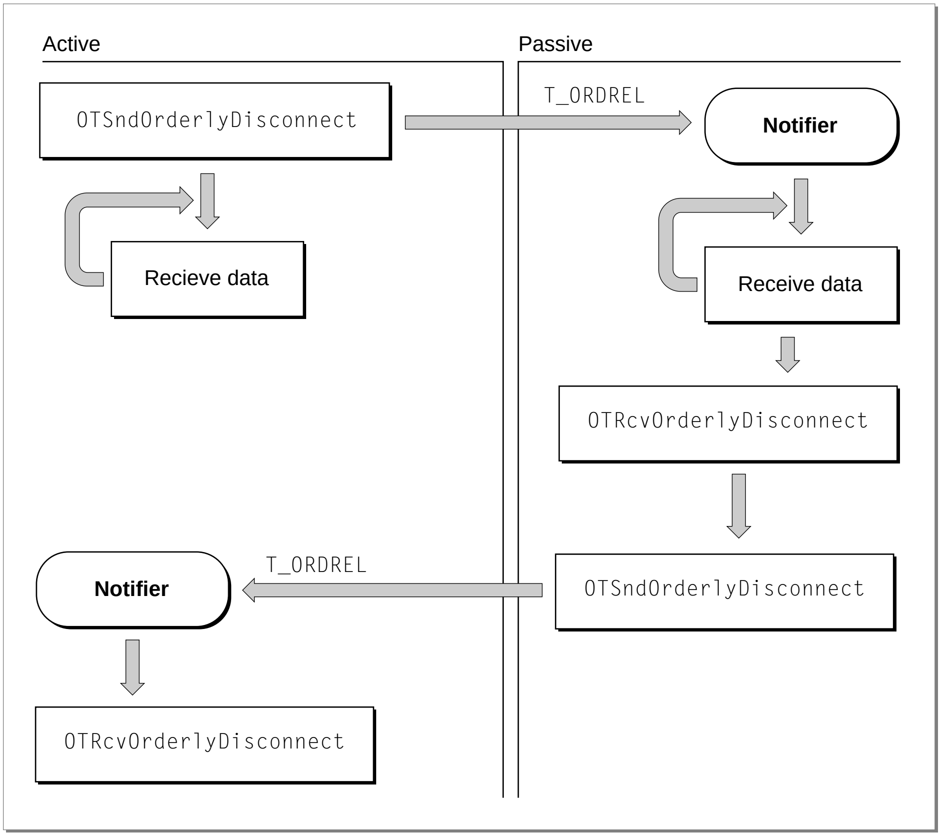
Figure 4-6 Remote orderly disconnect



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. (Once it calls 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 cannot read any more data; however, it can continue to send data. This is a *half-closed* connection. When the passive peer is finished sending any additional data, it calls 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 4-7 shows the steps required to complete a local orderly disconnect.

Figure 4-7 A local orderly disconnect



As you can see, the sequence of steps is very similar to that shown in Figure 4-6. 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 for all types of endpoints. After you read this section, you should read "Transferring Data Between Transactionless Endpoints" (page 119) or "Transferring Data Between Transaction-Based Endpoints" (page 121) for additional information about the type of endpoint you are using.

The chapter "Advanced Topics(page 215)" presents additional material that concerns the transfer of data and improving performance; this material includes

- sending non-contiguous data
- transferring data in raw mode
- doing no-copy receives

Please consult that chapter for more information.

Sending Data Using Multiple Sends

If you are sending a single 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 send function. This lets the provider know that it has not yet received the entire data unit.
2. Clear the `T_MORE` bit the last time you call the send 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.

IMPORTANT

Connectionless transactionless protocols do not support the `T_MORE` flag. ▲

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 need to call the receive function again and read more data until the T_MORE bit is cleared, which indicates that you have read the entire data unit.

Transferring Data Between Transactionless Endpoints

Open Transport defines two sets of functions that you can use to send and receive data between transactionless endpoints. 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 DDP and UDP, to send and receive discrete data packets.

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.

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 might issue 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 and TCP, 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 buffer where the data should be copied, the size of the buffer, and a `flags` value that Open Transport sets to tell the client whether 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. In general, it is recommended that you do not use expedited data because doing so results in code that is less 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 two ways of breaking up a data stream into logical size units.

- If the endpoint supports it, enable the use of the T_MORE flag bit to the OTSnd function. Then, 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 this flag bit.
- Use the data transferred with your first send to specify the name and size of the data that you want to send. The receiving endpoint can save the size value and decrement it as it receives bytes until the number equals 0. This last method is the only one that is transport-independent.

Transferring Data Between Transaction-Based Endpoints

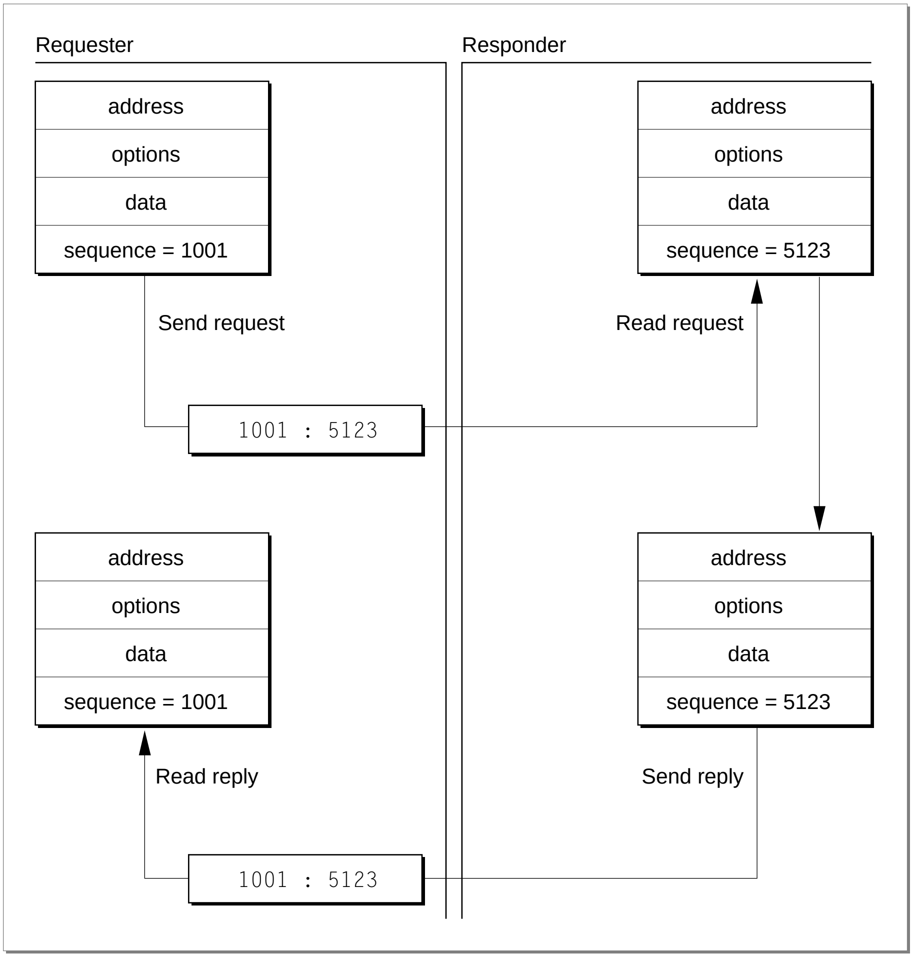
Open Transport defines two sets of functions that you can use to perform a transaction. One set is defined for connectionless transactions; the other set is defined for connection-oriented transactions. A **transaction** is a process during which one endpoint, the *requester*, sends a request for a service. 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 4-8 shows how the transaction ID is generated by the requesting application and the provider during the course of a transaction.

Figure 4-8 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 4-8, 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 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 (using the function `OTScheduleTimerTask`) 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.

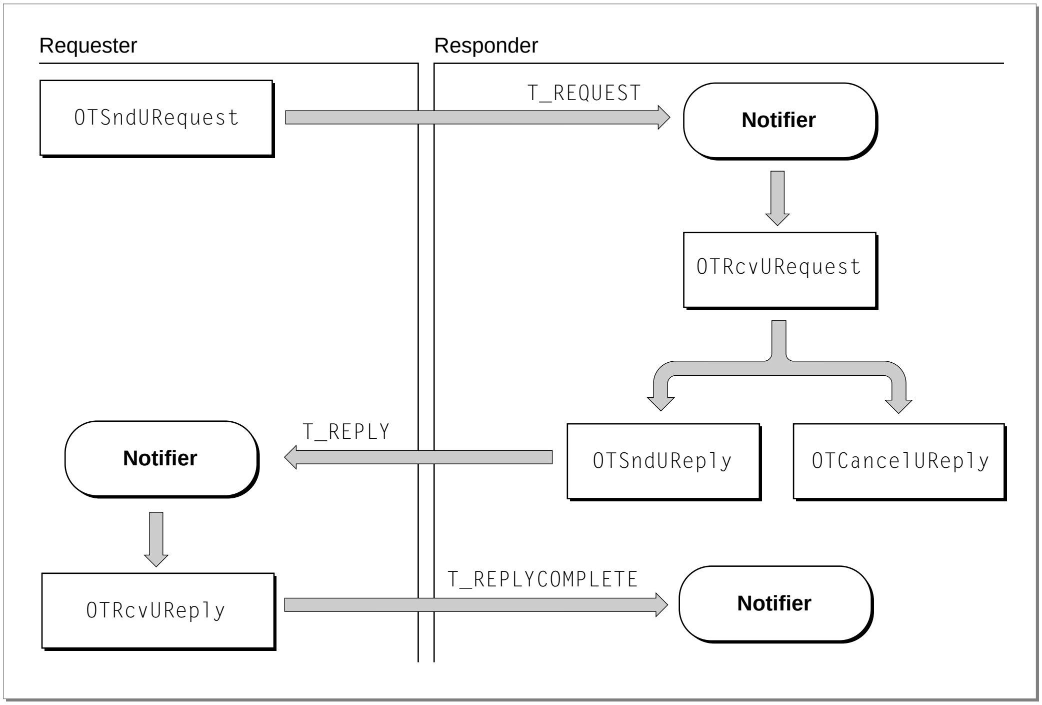
Having read the request, the responder can reply to the request using the OTSndUReply function or reject the request using the OTCancelUReply function. Although the requester is not advised that the responder has rejected a request, it's important that the responder explicitly cancel an incoming request 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 4-9 illustrates how connectionless transaction-based endpoints in asynchronous mode exchange data.

Figure 4-9 Data transfer using connectionless transaction-based endpoints in asynchronous mode

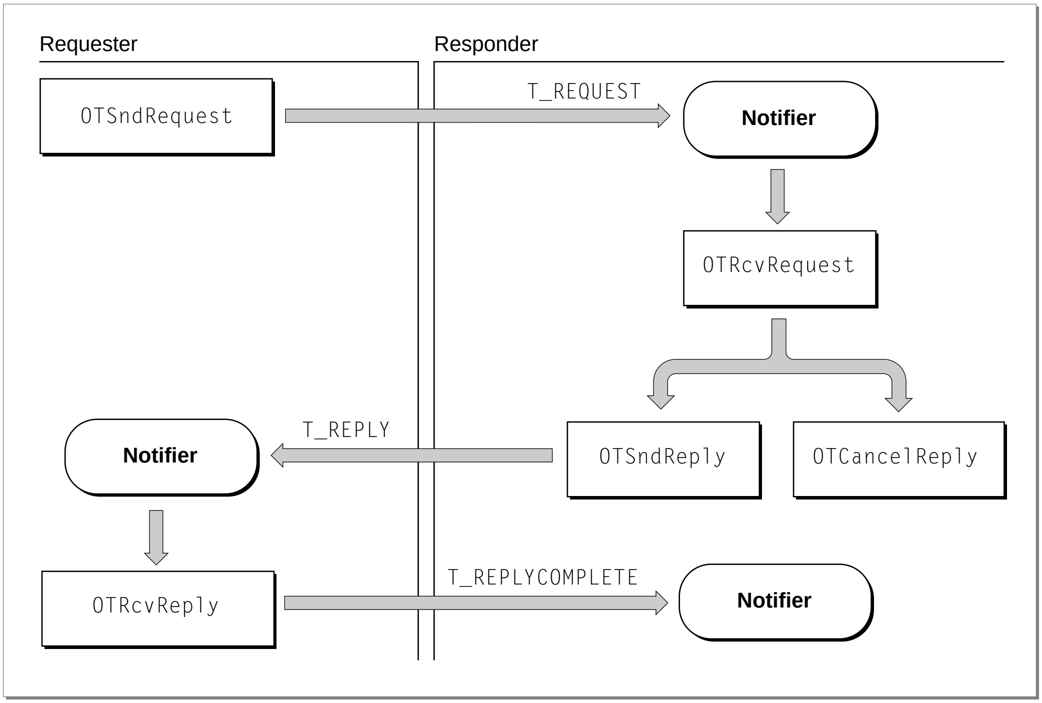


Using Connection-Oriented Transaction-Based Service

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" (page 123) describes the sequence of functions used to transfer data using a transaction. Figure 4-10 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 4-9 (page 125). Of course, you use different functions to complete these two types of transactions: the names of the functions shown in Figure 4-10 do not include a "U" in the function name.

Figure 4-10 Data transfer using connection-oriented transaction-based endpoints in asynchronous mode



For information about how to handle disconnection requests that might occur during a transaction, see "Using Orderly Disconnects" (page 114).

CHAPTER 5

Programming With Open Transport

Contents

This chapter examines methods of structuring Open Transport programs and discusses the relative merits of these methods in the context of the Mac OS cooperative multitasking environment. The chapter also takes a closer look at Mac OS interrupt levels and explains how Open Transport processing is affected by interactions between code executing at these levels. You should read this chapter if you are writing a server application, encountering synchronization problems, or want to improve performance. The chapter "Programming With Open Transport Reference" includes detailed information about the functions introduced in this chapter.

To use this chapter, you need to be familiar with system tasks, deferred tasks, and interrupts in general. For additional information about system tasks, read the information about the `SystemTask` function in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*, and for additional information about interrupts and deferred tasks, read the chapters "Introduction to Processes and Tasks" and "Deferred Task Manager" in *Inside Macintosh: Processes*.

Open Transport Programming Models

Designing a program that uses Open Transport involves finding an execution path that is simple to code but that does not degrade user experience nor endanger the robustness of your program. This section describes various strategies that you can use to structure code that calls Open Transport, focusing on the relative merits of the Open Transport notification mechanisms.

The Mac OS Open Transport API is a superset of the industry standard X/Open Transport Interface (XTI) specification. Because the XTI standard originated in a preemptive multitasking environment, a task's blocking I/O requests did not degrade the system's overall responsiveness. In such an environment all calls can be made synchronously, which eases the task of coding and minimizes synchronization problems. The matter stands differently in the current Mac OS cooperative multitasking environment, in which it is each task's responsibility to provide other, concurrent tasks with access to the processor. In the Mac OS environment, calling a task synchronously, without ceding time to other processes, is regarded as very poor programming practice and can easily hang the machine or seriously degrade user experience. To solve this problem, Open Transport extends the XTI API to support asynchronous notification of I/O completion. Open Transport uses several types of events to notify your application that something has occurred that requires its immediate attention. An event might signal the arrival of data, a connection or disconnection request, or the completion of an asynchronous function. Your program can either poll for these events or it can install a notifier function that Open Transport will call when an event occurs.

There are three basic ways to structure Open Transport programs:

- Synchronous processing with threads

  Using this method, you can call Open Transport functions synchronously. Open Transport sends your notifier the event `kOTSyncIdleEvent` whenever a synchronous call is waiting to complete. In response, your notifier can call the function `YieldToAnyThread`, which allows other concurrent processes to obtain processing time. This method offers the simplest programming model inasmuch as it avoids asynchronous processing. For more information, see "Using Synchronous Processing With Threads" (page 130).

- Polling for events

  Using this method, you can call the function `OTLook` from your main event loop to poll for events such as the arrival of data, connection and disconnection requests, etc. The problem with this method is that the `OTLook` function does not return completion events. Thus, if you are calling a function asynchronously, you need to find some other way to determine whether the function has completed. For more information, see "Polling for Events" (page 132).

- Using a notifier function to handle events

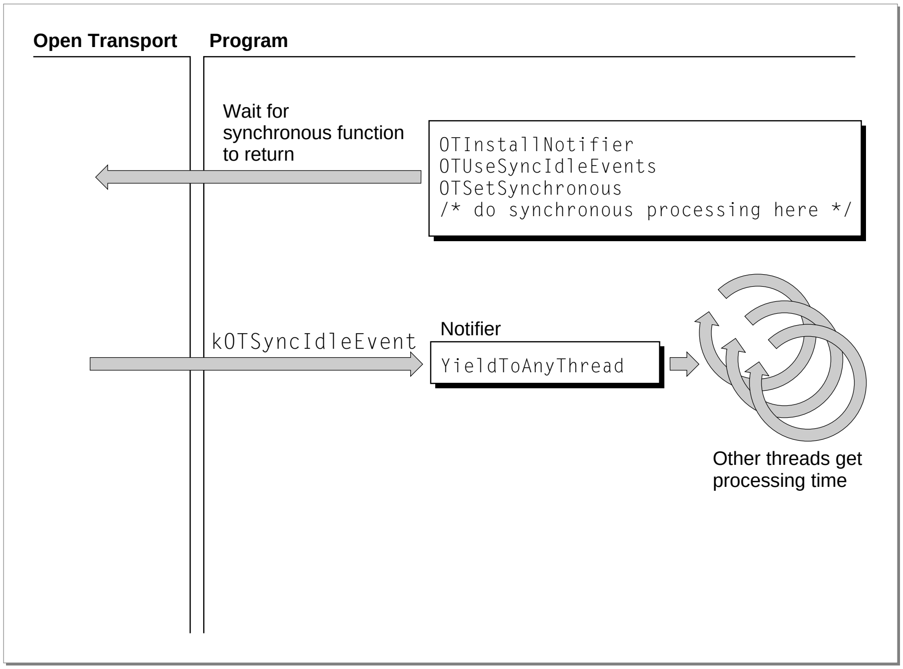
  Using this method, you install a notifier function and call Open Transport functions asynchronously. Open Transport sends you any events that affect the specified endpoint, and you handle these from your notifier or from your main event loop. This method offers the best performance, but it increases program complexity and might give rise to synchronization problems. For more information, see "Using Asynchronous Processing With a Notifier" (page 134).

Using Synchronous Processing With Threads

Figure 5-1 shows the key functions that your program must call to implement synchronous processing with threads. From within your program you must install a notification routine that handles the event `kOTSyncIdleEvent` by calling the function `YieldToAnyThread`. The program must also call the function

OTUseSyncIdleEvents to let Open Transport know that it wants to receive events of the type kOTSyncIdleEvent.

Figure 5-1 Synchronous processing with threads



When Open Transport is waiting for a synchronous function to complete, it sends the event kOTSyncIdleEvent to your notifier when it is safe for the notifier to call the function YieldToAnyThread. This function eventually causes the Thread Manager to switch to a thread that calls WaitNextEvent, thus yielding time to other processes.

Note
You must be familiar with the Thread Manager in order to use the YieldToAnyThread function. ◆

The only disadvantage of this method is that once you give time to other processes, you have no control over how long it takes for these processes to call

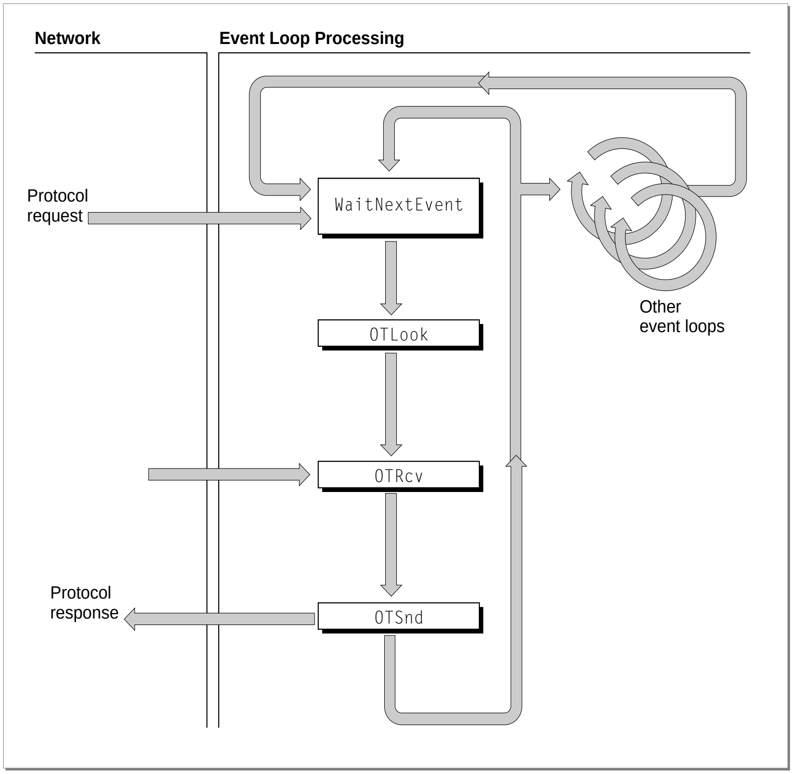
WaitNextEvent. So, while synchronous processing with threads might not be the method of choice for high performance servers, if your needs are more modest, you can enjoy the relative programming simplicity of this method. For a detailed example of a sample program using this model, see Listing 1-4 in "Getting Started With Open Transport(page 31)."

Note
To get out of a threaded synchronous routine, use the function OTCancelSynchronousCall (page 397).

Polling for Events

Figure 5-2 shows the structure of a program that calls Open Transport functions asynchronously and uses the OTLook function to poll for incoming events.

Figure 5-2 Polling for events

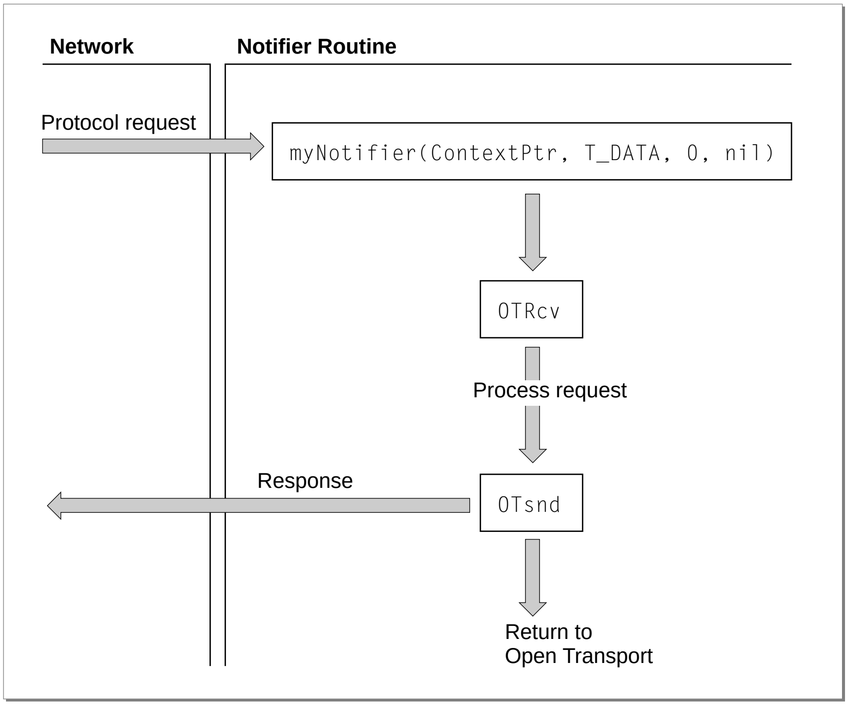


By using the `OTLook` function within its main event loop, an application does not need to idle while waiting for data to arrive. However, processing Open Transport events in an application's event loop can result in unpredictable packet processing delays. This is because the time between when your application receives a packet and when it responds depends on factors external to your application; it depends upon how other concurrent processes are using (or abusing) their access to the processor. Moreover, the `OTLook` function was written for the original XTI environment in which asynchronous processing played a very minor part. For this reason, the function does not return asynchronous completion events; as a result, if you are calling Open Transport functions asynchronously, you must use some other means to determine whether these have completed.

Using Asynchronous Processing With a Notifier

Figure 5-3 shows the structure of an application that calls Open Transport functions asynchronously and uses a notification routine to process asynchronous and completion events. The chapter "Providers(page 61)" gives detailed information about the use of notifiers.

Figure 5-3 Asynchronous processing with a notifier



As shown in the figure, the advantage of using the notifier is that it is called by Open Transport whenever an event occurs, allowing you to respond immediately. Because Open Transport often calls your notifier at deferred task time, you can handle requests without the overhead of event loop processing.

To get the best performance and to minimize synchronization problems, you should attempt to respond to most events directly in the notifier. You should be able to perform the following tasks from your notifier:

- accept and hand off connections
- receive and process all incoming data
- start asynchronous I/O operations; for example, call File Manager functions
- send network data
- tear down network connections

By the same token, because notifiers do often execute at deferred task time, they are somewhat limited in the functions they can call. For more information, see "Deferred Task Level" (page 137).

The following guidelines can help you use notifiers safely and effectively:

- Treat the notifier code path as a critical section. Assume you are locking the operating system from other tasks.
- Never call Open Transport at hardware interrupt time, except to schedule a deferred task or to call one of the functions (listed in Appendix C) that are safe to call at hardware interrupt time.
- Never make a synchronous Open Transport call from inside a notifier. Doing this will cause Open Transport to return the result `kOTStateChangeErr` in order to prevent you from deadlocking.
- Never make a synchronous File Manager or Device Manager call from inside a notifier. It might cause deadlock.
- Use completion events to gate endpoint action. For example, respond to a `T_OPENCOMPLETE` event by initiating a bind; or respond to the event `T_DISCONNECTCOMPLETE` by calling the `OTUnbind` function. Making such use of completion events will prevent you from receiving the result `kOTStateChangeErr` when you call a function before the endpoint is in a valid state.

Note
Note that Open Transport 68000-based applications can implement handler routines that use global variables without having to set up an A5 world. ◆

Interrupt-Safe Functions

One reason it's difficult to process packets in a notifier is that when you do, you can't call the Mac OS Toolbox functions that move memory at deferred task time. To remedy this, Open Transport makes available a number of fast and interrupt-safe utility functions that you can use instead. These functions are documented in the chapter "Utilities Reference" (page 621)."

Memory Management From Notifiers

You can safely call the functions `OTAllocMem` and `OTFreeMem` from your notifier. However, keep in mind that the memory allocated by `OTAllocMem` comes from the application's memory pool, which, due to Memory Manager constraints, can only be replenished at system task time. Therefore, if you allocate memory at hardware interrupt level or deferred task level, be prepared to handle a failure as a result of a temporarily depleted memory pool.

Interrupt Levels and Open Transport Processing

The Open Transport API offers a set of functions that you can use to schedule code to run at system task level, at deferred task level, and, in some cases, at hardware interrupt level. This section briefly describes the Mac OS interrupt levels, lists the restrictions on code executing at each level, and explains how you should use the functions provided by Open Transport to schedule code to run at these levels. This information is important in understanding the synchronization problems that might arise during asynchronous processing—where interaction between code executing at different levels might cause unexpected behavior.

Hardware Interrupt Level

Hardware interrupt-level execution happens as a result of a hardware interrupt request. Installable interrupt handlers for PCI bus, NuBus and other devices, as well as interrupt handlers supplied by Apple all execute at this level.

In general, you should minimize the amount of time that your code spends executing at hardware interrupt level. If you think you need to do extended processing at this level, you should consider trying to defer such processing to deferred task level. For information about how you do this, see "Deferred Task Level" (page 137).

If virtual memory is on, paging is not safe at hardware interrupt level unless the interrupt has been postponed using the Memory Management function `DeferUserfn`. Some system interrupt handlers (Device Manager completion routines, VBLs, slot VBLs, Time Manager tasks) automatically defer their operation to a safe time, but other hardware interrupt handlers must be sure not to cause page faults.

Open Transport furnishes a number of utility functions that you are allowed to call at hardware interrupt time. Appendix C(page 793) lists these functions. In some cases, you must notify Open Transport that you are about to call an Open Transport function at hardware interrupt time by first calling the `OTEnterInterrupt` function. You can then call one of the permitted functions. When you are done with calling Open Transport functions at hardware interrupt time, you must call the `OTLeaveInterrupt` function. For example, you could execute these code statements in this sequence:

```
OTEnterInterrupt();
OTScheduleDeferredTask(dtCookie);
OTLeaveInterrupt();
```

▲ **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 when these are required, you will either get the `OTBadSyncErr` result code or crash your system, depending on the function you call. ▲

Deferred Task Level

A deferred task is the means whereby you can schedule a routine to be executed from hardware interrupt level code. Deferred task processing occurs just before the operating system returns from hardware interrupt level to system task time. Scheduling code to run at deferred task time minimizes the time that code executes at hardware interrupt level and therefore minimizes system interrupt latency. Deferred tasks are executed serially, offering a simple mutual exclusion mechanism.

Programs using Open Transport can cause code to run at deferred task time by creating a deferred task with the `OTCreateDeferredTask` function and by scheduling it to run using the `OTScheduleDeferredTask` function. Using Open Transport functions to create and schedule deferred tasks is preferable to using the Deferred Task Manager function `DTInstall`, because by doing so you allow Open Transport to adapt to changes in the underlying operating system without having to change your code.

Code also executes at deferred task time if it is called by something that is executing at deferred task level. For example, Open Transport often calls notifier functions at deferred task level. You should assume, in writing your notifier functions, that they are likely to run at deferred task level and observe the restrictions on code running at this level.

If you are writing a system extension or a code resource, you probably need to use Open Transport's deferred task functions to get processing time to handle such tasks as allocating memory or accessing disk space. You must schedule a deferred task if you want to call such code from code that executes at interrupt time or from within an interrupt function such as a Time Manager function, Vertical Retrace Manager function, File Manager completion routine, or Device Manager completion routine.

Virtual Memory paging is safe at deferred task level. You can also call many Open Transport functions at deferred task time; these functions are described in Appendix C(page 793).

IMPORTANT

If you are writing a PCI device driver (`ndrv`), please note that Open Transport treats secondary interrupt level as hardware interrupt level. Therefore, your secondary interrupt handler is subject to the same restrictions as code running at hardware interrupt time, as described in the previous section. ▲

System Task Level

System task level is the level at which most application code executes. An application's main entry point is called at system task level. Cooperatively scheduled Thread Manager threads also run at system task time.

Open Transport furnishes several functions that you can use to schedule code to execute at system task level. Normally, you don't need to use these functions because your application executes within a normal event loop that runs at system task level. However, you might want to use Open Transport's system task scheduling functions for some of your application's processing because these functions provide an efficient way to streamline your main event loop. For example, you can avoid 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.

System task level is not considered interrupt level by any part of the system. Consequently, you can call anything at system task level. Virtual Memory paging is also safe at this level unless your code accesses some resource that the system needs to support paging. For example, if you get exclusive access to the SCSI bus by calling the function `SCSIGet`, you must not cause a page fault even at system task level.

Using Timer Tasks

Open Transport provides functions that you can use to create a timer task, to schedule the task, to cancel it, and to dispose of it. These functions are described in "Working With Timer Tasks" (page 529). Open Transport executes timer tasks at deferred task time.

IMPORTANT

You cannot call these functions from 68000 code running on a Power PC. ▲

Using System and Deferred Tasks

You can use Open Transport functions to schedule a callback function that will be called at system task time or deferred task time. To do this, you use the function `OTCreateSystemTask` or the function `OTCreateDeferredTask` to create the task. Then you use the function `OTScheduleSystemTask` or the function `OTScheduleDeferredTask` to schedule 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 when scheduling, cancelling, or destroying the task. When you create the task, you can also specify user-defined context information that Open Transport will pass to your task when it calls it. For 680x0 code, Open Transport also restores the A5 world to what it was when you created the task.

Calling Open Transport Functions

Appendix C(page 793) includes a table that lists all the Open Transport functions you can call at deferred task time. In general, you can make all endpoint calls from a deferred task as long as the endpoint is in asynchronous mode. A select number of Open Transport calls can only be made at system task time.

IMPORTANT

Because opening the first endpoint for a configuration requires that Open Transport load libraries, doing this from a deferred task will only work if the foreground task is calling the functions `WaitNextEvent`, `GetNextEvent`, or `SystemTask`. Subsequent asynchronous open calls from a deferred task will work regardless of what the foreground task is doing because the libraries will have already been loaded. ▲

Scheduling Tasks

Once you have created a task, you need to schedule it for execution. To do this, you use the functions `OTScheduleSystemTask`, `OTScheduleDeferredTask`, or `OTScheduleInterruptTask`. You pass the task reference (using the `stCookie` or the `dtCookie` parameter) to the function, and Open Transport attempts to schedule the task. If a system task is scheduled successfully, it executes when the `SystemTask` function next executes. If a deferred task is scheduled successfully, it executes as soon as possible after hardware interrupts have finished executing.

Because a system task can happen relatively slowly, enough time can elapse between scheduling and execution to let you cancel the task before it runs. If you use the `OTCancelSystemTask` function, you notify Open Transport not to execute the system task at the scheduled time. The reference remains valid, and you can choose to reschedule the task by using the `OTScheduleSystemTask` function again at any time. Deferred tasks, however, typically execute too quickly to allow time for canceling them.

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. This means that exactly the same task executes, which is useful for repetitive periodic tasks.

Deallocating Resources

You can destroy a task with the `OTDestroySystemTask` or the `OTDestroyDeferredTask` functions. 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 want to use a task after you have destroyed it, you must begin again by creating a new task with the `OTCreateSystemTask` or the `OTCreateDeferredTask` functions.

Handling Synchronization Problems

If you call certain Open Transport functions from different interrupt levels, synchronization problems can occur. For example,

1. You call the function `OTRcv` from your main thread.
2. There is no pending data; just as the function is about to return to the application with the result `kOTNoDataErr`, an inbound data packet interrupts Open Transport, and it steps up to deferred task time to process the data.
3. Open Transport calls your notifier with a `T_DATA` event, which you ignore because you are not aware of the possibility that the execution of the `OTRcv` function could be interrupted by the actual arrival of data (processed in a different interrupt context).
4. The call to `OTRcv` in your main thread completes with the result `kOTNoDataErr`, you have no way of knowing that you got the `T_DATA` event, and you won't get another one until you call the function again, for another `kOTNoDataErr` result. Consequently, your application hangs.

The solution to this problem is to adopt a sensible synchronization model—that is, do everything in your notifier (using the `OTEnterNotifier` function when you can't) or do everything at system task time. The key is not to mix and match execution levels for the same endpoint.

Handling Multiple Simultaneous Connections

This section describes the problems of handling multiple simultaneous connections and explains the use of the `tilisten` module as a means of handling these problems.

Problems With Accepting Multiple Simultaneous Connections

One of the big challenges of programming Open Transport is accepting multiple simultaneous incoming connections. The problem is that the obvious code stream can produce unexpected results. Take, for example, the following sequence:

1. You have a listening endpoint (one bound with a qlen greater than 0)in asynchronous mode. (The problem is independent of the mode of the listening endpoint but, for the sake of this example, we'll assume the listening endpoint is in asynchronous mode.)
2. An incoming connection arrives, and the listening endpoint calls your notifier with a `T_LISTEN` event.
3. Your notifier reads the details of the incoming connection using the `OTListen` routine.
4. Your notifier decides to accept the incoming connection by calling the function `OTAccept`.
5. However, the OTAccept call fails with a `kOTLookErr` because there is another pending `T_LISTEN` event on the listening endpoint. (This behavior is explicitly allowed in the XTI specification.)

There are a number of ways to solve this problem. The easiest approach is to use a qlen of 1 when you bind the endpoint. If you do this, the provider will reject connection attempts while you are in the process of accepting a connection attempt. The drawback to this approach is that the remote peer will receive unnecessary connection rejections.

A second approach is to program around the problem using the existing Open Transport APIs. When you call `OTAccept` and get a look error, you turn around and call `OTLook`. If the pending event is a `T_LISTEN`, put the first connection on hold and deal with this new connection. Of course, the attempt to accept the new connection can also fail because of a pending T_LISTEN, which means you have to put this second connection attempt on hold, and so on. This requires you to have a queue of pending incoming connection attempts. You must also deal with T_DISCONNECT events on the listening endpoint, and delete the corresponding connection attempts from the queue.

This second approach requires a lot of code, and is very hard to get right.

The third, and recommended, approach is to use the tilisten module to serialize incoming connection requests. This approach is described in the next section.

Using "tilisten" to Accept Multiple Simultaneous Connections

The tilisten module is provided to simplify the job of accepting multiple simultaneous incoming connections. The module sits on top of the provider associated with the listening endpoint and monitors connection requests being sent up to the client. When a connection request arrives while the client is still in the process of dealing with an earlier connection request, the tilisten module holds on to the second connection request until the client is accepts or rejects the first one.

Thus, when the tilisten module is installed in the stream, the OTAccept function will never fail because of a pending T_LISTEN event, and only fail with a pending T_DISCONNECT event if that disconnection event is for the current connection request.

You can use the tilisten module in your listening endpoint by specifying it in the configuration you use to build the listening endpoint. For example, if you want to create an TCP endpoint with the "tilisten" module, you would do so with the following code:

```
ep = OTOpenEndpoint(OTCreateConfiguration("tilisten,tcp"), 0, nil, &err);
```

You should should only include the tilisten module in connection-oriented, listening endpoints. The module is not appropriate for use in hand-off endpoints, or endpoints used for outgoing connections.

Note
The tilisten module is not available in versions of Open Transport prior to version 1.1.1. ◆

Improving Performance

The following suggestions for improving performance were drawn up with servers in mind; however, if your application needs to handle multiple connection requests, you might find the section “Streamlining Endpoint Creation” useful. For additional information on handling throughput to improve performance, see the chapter “Advanced Topics.”

Streamlining Endpoint Creation

The time required to create and open an endpoint can delay connection set-up time. This can adversely affect servers, especially HTTP servers, since they must manage high connection turnover rates. To handle this problem, follow these guidelines:

- Preallocate endpoints

  Preallocate a pool of open, unbound endpoints into an endpoint cache. When a connection is requested (you receive a `T_LISTEN` event), you can dequeue an endpoint from this cache and pass it to the function `OTAccept`.

  Using this method, the only time you have to wait for an endpoint to be created is if the queue is empty, when you must allocate an additional block of endpoints.

- Recycle endpoints

  You can use an endpoint-cache to recycle endpoints when your connection is closed. Rather than call the function `OTCloseProvider` each time a connection terminates, cache the unbound endpoint. This keeps it available for a subsequent open request.

  To use this method, unbind the endpoint upon receipt of the `T_DISCONNECT` event. Then, when the notifier receives the `T_UNBINDCOMPLETE` event, queue that endpoint into your endpoint cache. Optionally, to save memory, you can deallocate the endpoint when the endpoint cache reaches some predetermined limit.

- Create clone configurations

  Another way to speed up endpoint creation is to create a prototype configuration structure with the function `OTCreateConfiguration`. Then, use the OTCloneConfiguration function to pass the configuration structure to the function OTOpenEndpoint. The call to OTCloneConfiguration is about five times faster than that to the OTCreateConfiguration function.

Handling Dead Clients

A properly designed server should be prepared to handle what happens when a remote client unexpectedly disappears. This problem is further aggravated when the link has been flow-controlled. For example:

1. You are transmitting a large amount of data to a client.
2. Your transport provider enters a flow-control state.
3. The client crashes or becomes unreachable.
4. After a timeout, your server decides to force a disconnect from that client and issues a disconnect request.
5. However the T_DISCONNECT event is subject to flow control, which causes your link to hang.

You can solve this problem by flushing the stream before requesting the disconnection. The best way to do this is to send the I_FLUSH command to the stream head using the OTIoctl function. For example:

```
#include <stropts.h>

/* check to see if you are already disconnected */

error = OTIoctl(ep, I_FLUSH, (void*) FLUSHRW);
if error OTUnbind(ep)
    .....

MyNotifyProc (... void* the Param) {
    case kStreamIoctlEvent                  /* flush is complete */
        (void) OTSndDisconnect (ep, NULL);  /* safe to disconnect */
        break;
}
```

This will result in your notifier receiving all T_MEMORYRELEASED events for any outstanding send calls that acknowledge sends. You should then attempt to send the disconnection request.

Shutting Down Servers

To shut down an Open Transport network server properly, you need to:

- Make sure that all network and I/O operations have either completed or aborted.
- Flush any flow-controlled data streams with the `I_FLUSH` command. See "Handling Dead Clients" (page 145) for detailed information.
- Unbind and close all endpoints.
- Cancel any outstanding deferred tasks with the function `OTDestroyDeferredTask`.
- Release any `OTBuffer` structures with the function `OTReleaseBuffer`.
- Dispose of any unused configuration structures with the function `OTDestroyConfiguration`.

CHAPTER 6

Mappers

Contents

Mappers

This chapter describes mappers, a 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 want to connect to a remote endpoint simply by specifying its name, you can simply pass the name to the `OTConnect` function.

If you are using an endpoint that cannot be bound by name, if you want to look up a name pattern, if you want to register a name that is not associated with an endpoint, 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 “Mappers Reference” (page 545).

Mapper providers, like all Open Transport providers, can operate synchronously or asynchronously and can block. For general information about Open Transport providers, see the chapter “Providers” (page 61).

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. Which mapper functions you call 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 domain name resolver (DNR), you can only look up a name that has been registered using other means.

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 create a mapper that executes asynchronously by calling the `OTAsyncOpenMapper` function. 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.

Mappers use one asynchronous event and four completion events. Table 6-1 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 in "Functions" (page 550).

Table 6-1 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.

Table 6-1 Completion events for asynchronous mapper functions (continued)

Completion code	Meaning
`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 might be 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"(page 61) 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. 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.

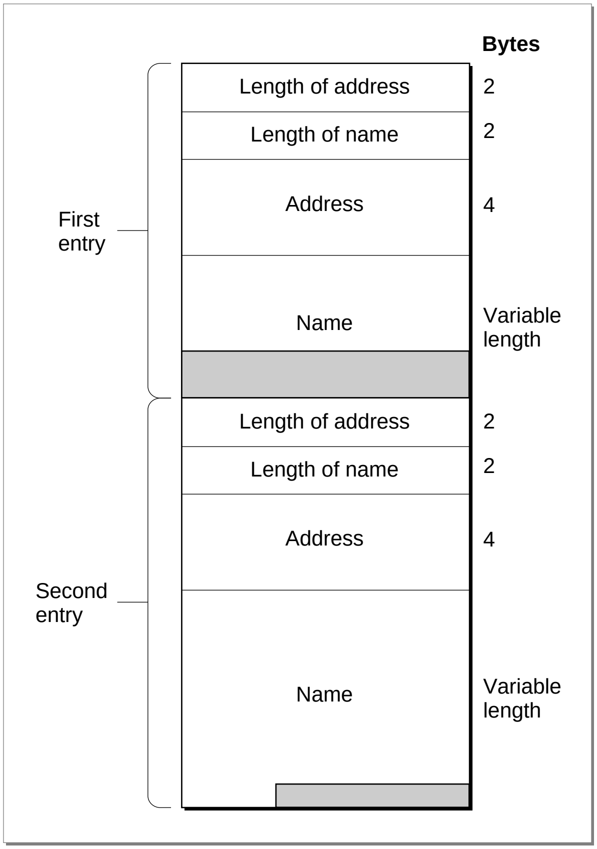
Searching for Names

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 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 the number you specify for the `maxcnt` field 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 6-1 shows how the contents of a reply buffer containing two entries are stored. The section “Code Sample: Using OTLookupName” (page 155) provides and describes a sample program that uses the `OTLookupName` function. See especially, Listing 6-3.

Figure 6-1 Format of entries in OTLookupName reply buffer



The first two bytes of each entry specify the length of the address; the second two bytes specify the length of the name. The address is stored next and then the name, padded to a four byte boundary.

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 sends two event codes: it sends the T_LKUPNAMERESULT code each time it stores a name in the reply buffer, and it sends 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. 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. You can also do it from code bracketed by the `OTEnterNotifier` and `OTLeaveNotifier` functions. For more information, see "OTEnterNotifier" (page 408).
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.

Note
The `T_LKUPNAMECOMPLETE` event might have stored a name in the buffer. Be sure to check for this possibility. ◆

Code Sample: Using OTLookupName

This section discusses the program OTLookupNameTest, which demonstrates how you open an NBP mapper provider, issue an NBP lookup request, and print out the resulting information. Listing 6-1 shows the preprocessor directives and the main function of the program.

Listing 6-1 The main function to OTLookupNameTest

```
#ifndef qDebug          /* variable set for OT debugging macros */
#define qDebug1
#endif

#include <OpenTransport.h>
#include <OpenTptAppleTalk.h>
#include <OTDebug.h> /* Need OTDebugBreak & OTAssert macros */
#include <stdio.h>

/* OTDebugStr is not defined in OT header files, but it is
exported by the libraries, so we define the prototype here. */
extern pascal void OTDebugStr(const char* str);

static UInt32 gLastPrinted = 0; /* Global var to track printing */

void main(void)
{   OSStatus    err;
    char        requestAddress[] = "=:AFPServer@*";

    printf("Hello World!\n");

    err = InitOpenTransport();
    if (err == noErr) {
        err = LookupAndPrint(requestAddress);
        CloseOpenTransport();
    }
    if (err == noErr) {
        printf("Success.\n");
    } else {
        printf("Failed with error %d.\n", err);
    }
    printf("Done.  Press command-Q to Quit.\n");
}
```

The main function initializes Open Transport, calls the user-defined function `LookupAndPrint` (passing a value for the requested address), and then closes Open Transport.

The LookupAndPrint function is the key function to the OTLookupNameTest program. However, because it calls Open Transport functions synchronously, it also uses a notifier to yield time to other processes. Listing 6-2 shows the notifier, which calls printf periodically in response to a kOTSyncIdle event. (The printf function calls WaitNextEvent, thus our synchronous calls to Open Transport will yield time to other processes. A real world application would probably use threads to do this.

Listing 6-2 Notifier that yields time to other processes

```
static pascal void YieldingNotifier(EndpointRef ep, OTEventCode code,
                                    OTResult result, void* cookie)
{
    #pragma unused(ep)
    #pragma unused(result)
    #pragma unused(cookie)

    switch (code) {
        case kOTSyncIdleEvent:
            if (TickCount() > gLastPrinted + 10) {
                printf(".");
                fflush(stdout);
                gLastPrinted = TickCount();
            }
            break;
        default:
            /* do nothing */
            break;
    }
}
```

For more information on using threads to yield time, see “Using Synchronous Processing With Threads” (page 130).

Listing 6-3 shows the LookupAndPrint function. This function takes one parameter, a pointer to an NBP address. This address must have the form

<name>:<type>@<zone>

The function begins by opening an NBP mapper provider and switching it into synchronous/blocking mode. It uses kOTSyncIdle events (and the notifier shown in Listing 6-2) to yield time to other processes. Then it issues an NBP lookup request, using the `OTLookUpName` function (page 559). When the request completes, the function calls the user-defined `PrintAddress` and `PrintName` functions to display the results.

Listing 6-3 The LookupAndPrint function

```
static OSStatus LookupAndPrint(char *requestAddress)
{
    OSStatus        err;
    OSStatus        junk;
    MapperRef       nbpMapper;
    TLookupRequest  lookupRequest;
    TLookupReply     lookupReply;
    UInt8           *responseBuffer;
    TLookupBuffer   *currentLookupReplyBuffer;
    UInt32           nameIndex;

    err = noErr;
    nbpMapper = kOTInvalidMapperRef; /* for error checking */

    /* Create the responseBuffer. */

    responseBuffer = OTAllocMem(kResponseBufferSize);
    if (responseBuffer == nil)
        err = kENOMEMErr;

    /* Create an NBP mapper and set it to up for threaded processing. */

    if (err == noErr)
        nbpMapper = OTOpenMapper(OTCreateConfiguration(kNBPName),
                                                    0, &err);
    if (err == noErr) {
        junk = OTSetSynchronous(nbpMapper);
        OTAssert("LookupAndPrint: Could not set synchronous mode
                                            on mapper", junk == noErr);
        junk = OTSetBlocking(nbpMapper);
        OTAssert("LookupAndPrint: Could not set blocking mode
                                        on mapper", junk == noErr);
        junk = OTUseSyncIdleEvents(nbpMapper, true);
        OTAssert("LookupAndPrint: Could not enable sync idle events
                                        on mapper", junk == noErr);
        junk = OTInstallNotifier(nbpMapper, YieldingNotifier, nil);
        OTAssert("LookupAndPrint: Could not install notifier
                                        for mapper", junk == noErr);
    }

    /* Call OTLookupName synchronously. */

    if (err == noErr) {

        /* Set up the TLookupRequest structure. */

        OTMemzero(&lookupRequest, sizeof(lookupRequest));
        lookupRequest.name.buf = (UInt8 *) requestAddress;
        lookupRequest.name.len = OTStrLength(requestAddress);
        lookupRequest.timeout = 1000;// 1 second in milliseconds
        lookupRequest.maxcnt = kResponseBufferSize /
                                   kNBPEntityBufferSize;

        /* Set up the TLookupReply structure. */

        OTMemzero(&lookupReply, sizeof(lookupReply));
        lookupReply.names.buf = responseBuffer;
        lookupReply.names.maxlen = kResponseBufferSize;

        /* Now do the lookup. */

        err = OTLookupName(nbpMapper, &lookupRequest, &lookupReply);
    }

    /* Print out the contents of the responseBuffer. */

    if (err == noErr) {
        printf("\n");

        /* Start by pointing to the beginning of the response buffer. */

        currentLookupReplyBuffer = (TLookupBuffer *) responseBuffer;

        /* For each response in the buffer... */
        for (nameIndex = 0; nameIndex < lookupReply.rspcount;
                                    nameIndex++) {

        /* ... print the name and address and... */

            printf("%3d ", nameIndex);
            PrintAddress( (DDPAddress *)
                        &currentLookupReplyBuffer->fAddressBuffer[0]);
            PrintName( (char *)&currentLookupReplyBuffer->
                        fAddressBuffer[currentLookupReplyBuffer->
                                        fAddressLength],
                            currentLookupReplyBuffer->fNameLength);
            printf("\n");

            /* ... use OTNextLookupBuffer to get from the current
                                    buffer to the next. */

            currentLookupReplyBuffer =
                    OTNextLookupBuffer(currentLookupReplyBuffer);
        }
    }

    /* Clean up. */

    if (responseBuffer != nil) {
        OTFreeMem(responseBuffer);
    }
    if (nbpMapper != kOTInvalidMapperRef) {
        junk = OTCloseProvider(nbpMapper);
        OTAssert("LookupAndPrint: Failed closing mapper", junk == noErr);
    }

    return err;
}
```

The function `LookupAndPrint` calls two functions, `PrintName` and `PrintAddress`, to print names and addresses; Listing 6-4 shows the two functions.

Listing 6-4 Printing names and addresses

```
static void PrintName(const char *name, UInt32 length)
{
    char nameForPrinting[256];

    OTMemzero(nameForPrinting, 256);
    OTMemcpy(nameForPrinting, name, length);

    printf("“%s”", nameForPrinting);
}

static void PrintAddress( DDPAddress *addr )
{
    OTAssert( "PrintAddress: Expected a DDPNBPADdress",
                        addr->fAddressType == AF_ATALK_DDP );
    printf("Net = $%04x, Node = $%02x, Socket = $%02x ",
                addr->fNetwork,
                addr->fNodeID,
                addr->fSocket);
}
```

CHAPTER 7

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 degrades transport independence. 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 how you construct an options buffer, how you get and set option values, and how you verify values. It also provides code samples that show how you

- construct option buffers
- parse buffers containing option information
- get, set, and display option values.

To understand this chapter, you should be familiar with endpoint providers and the endpoint functions used to transfer data. These topics are discussed in “Endpoints”(page 83). For specific information about the options that are supported for a protocol implementation, you need to consult the documentation provided for that protocol.

In general, it is recommended that you set up your options using the OTOptionManagement function after creating your endpoint and before using it. The actual semantics of option negotiation are somewhat complicated and are covered in “XTI Option Summary” (page 807).

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.

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.

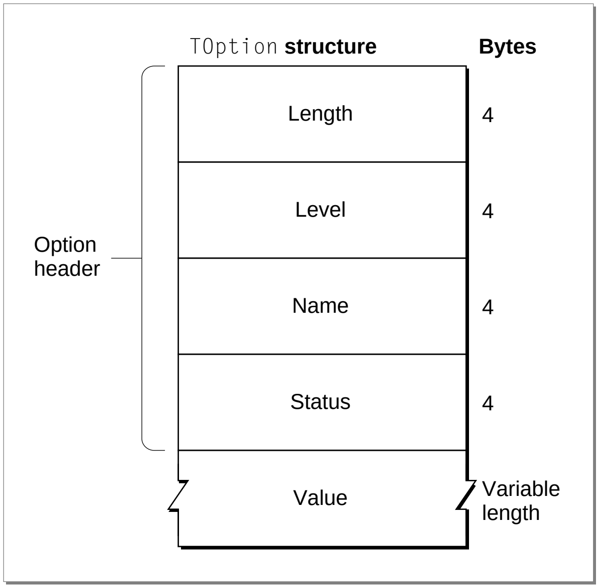
Types of Options

The process of option negotiation is affected by the type of option involved. Options can be association-related, privileged, read-only, or absolute. For more information about these distinction and how they affect option negotiation, see “XTI Option Summary” (page 807).

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 7-1 shows, an option is described by an option header and a value.

Figure 7-1 The format of option information



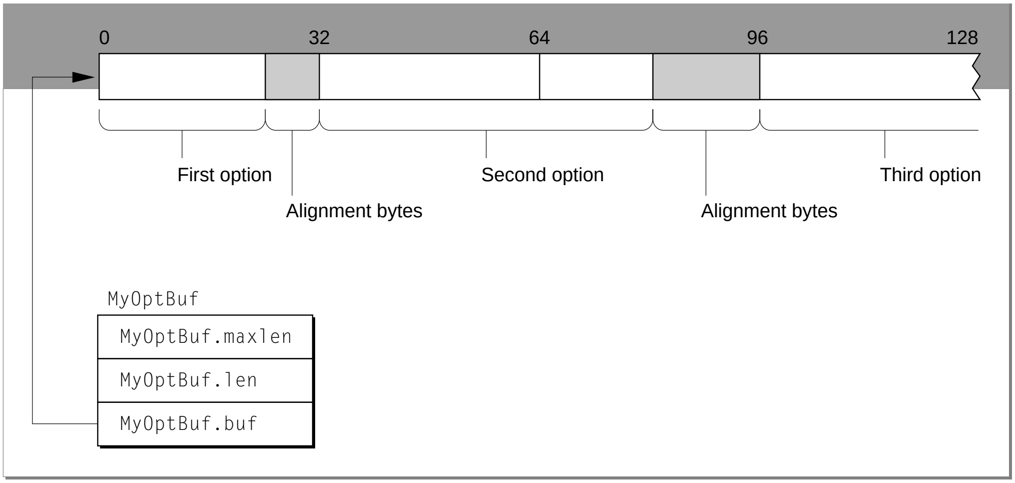
The option header is the same for all options. It contains four fields that specify:

- 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 7-2 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 7-2 An options buffer



You can concatenate several TOption structures in a buffer, as shown in Figure 7-2, provided you observe the following rules:

- TOption structures must be quad-byte 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 7-1 provides a brief summary of XTI-level options. For more detailed information about these options, see "XTI-Level Options"(page 565).

Table 7-1 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 number of bytes that must 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 7-2. 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"(page 567).

Table 7-2 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 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.

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 basic distinction between setting option values using the

OTOptionManagement function and using any of the other endpoint functions is that options negotiated with the OTOptionManagement function affect all functions called by an endpoint, whereas options negotiated using any other function affect only the connection, transaction, or datagram for which they are set. For more detailed information about these differences, see "XTI Option Summary" (page 807).

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. Appendix D 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" (page 816) 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 7-1 (page 168). You can concatenate several structures in the buffer, as shown by Figure 7-2 (page 169), 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.

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 7-1 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 7-1 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.

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" (page 172) 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.

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 7-2 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 7-2 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.

Sample Code: Getting and Setting Options

The code listings discussed in this section furnish examples of how you can use the Open Transport API to get, set, and display the values of options.

Listing 7-3 shows a main function that calls a number of other functions (defined in subsequent listings) to set, get, and display option values.

Listing 7-3 Calling functions that get, set, and display options

```
#ifndef qDebug      / *OT debugging macros need this var */
#define qDebug 1
#endif
#include <OpenTransport.h>

#include <OpenTptInternet.h> /* for TCP/IP */

#include <OpenTptSerial.h>  /* for serial endpoints */

#include <OTDebug.h>    /* Need OTDebugBreak and OTAssert macros */
#include <stdio.h>  /* Standard C prototypes */

/* OTDebugStr is not defined in any OT header files, but it is
exported by the libraries, so we define the prototype here. */

extern pascal void OTDebugStr(const char* str);

void main(void) {
    OSStatus err;
    OSStatus junk;
    EndpointRef ep;
    UInt32 value;

    printf("HelloWorld!\n");

    err = InitOpenTransport();

    if (err == noErr) {
            ep = OTOpenEndpoint(OTCreateConfiguration(kTCPName), 0, nil,
                                                            &err);
    if (err == noErr) {
      printf("\nGetting and Setting IP_REUSEADDR.\n");
      err = GetFourByteOption(ep, INET_IP, IP_REUSEADDR, &value);
      if (err == noErr)
        printf("Default value = %d\n", value);
      if (err == noErr)
        err = SetFourByteOption(ep, INET_IP, IP_REUSEADDR, true);
      if (err == noErr){
        err = GetFourByteOption(ep, INET_IP, IP_REUSEADDR,
                                                    &value);
        if (err == noErr)
          printf("New value = %d\n", value)
      }
      if (err == noErr) {
        printf("\nPrinting Options Piecemeal at Level
                                                    INET_IP.\n");
        err = PrintAllOptionsAtLevel(ep, INET_IP);
      }
      if (err == noErr){
        printf("\nPrinting Formatted Options at Level
                                                  COM_SERIAL.\n");
        err = PrintOptionsForConfiguration(kSerialName,
                                                    COM_SERIAL);
      }
      if (err == noErr) {
        printf("\nBuilding Options for COM_SERIAL.\n");
        err = BuildAndPrintOptions(kSerialName, "BaudRate=9600,
                                        DataBits=7, StopBits=15");
      }
      junk = OTCloseProvider(ep);
      OTAssert("GetSetOptions: Closing the endpoint failed",
                                                junk == noErr);
    }
    CloseOpenTransport();
  }
  if (err == noErr)
    printf("Success.\n");
   else
    printf("Failed with error %d.\n", err);
  printf("Done.  Press command-Q to Quit.\n");
}
```

This main function initializes Open Transport and then creates a TCP endpoint. Then, it calls the function GetFourByteOption to obtain the value of the IP_REUSEADDR option, which governs whether you can bind multiple endpoints to addresses with the same port number. Listing 7-4 contains the definition of the GetFourByteOption function.

Listing 7-4 Getting an option value

```
static OTResult GetFourByteOption(EndpointRef ep,
                                   OTXTILevel level,
                                   OTXTIName  name,
                                   UInt32   *value)
{
    OTResult    err;
    TOption     option;
    TOptMgmt    request;
    TOptMgmt    result;

    /* Set up the option buffer */
    option.len      = kOTFourByteOptionSize;
    option.level    = level;
    option.name     = name;
    option.status   = 0;
    option.value[0] = 0;// Ignored because we're getting the value.

    /* Set up the request parameter for OTOptionManagement to point
     to the option buffer we just filled out */

    request.opt.buf = (UInt8 *) &option;
    request.opt.len = sizeof(option);
    request.flags   = T_CURRENT;

    /* Set up the reply parameter for OTOptionManagement. */
    result.opt.buf       = (UInt8 *) &option;
    result.opt.maxlen    = sizeof(option);

    err = OTOptionManagement(ep, &request, &result);

    if (err == noErr) {
        switch (option.status)
        {
            case T_SUCCESS:
            case T_READONLY:
                *value = option.value[0];
                break;
            default:
                err = option.status;
                break;
        }
    }

    return (err);
}
```

The function GetFourByteOption gets the current option setting and assigns it to the location referenced by value. The endpoint is assumed to be in synchronous mode. If an error occurs, the function returns a negative result. If the option could not be read, a positive result (either T_FAILURE, T_PARTSUCCESS, or T_NOTSUPPORT) is returned.

Within the body of the function, the fields of the option buffer are set up to specify the option and value we want to get. The TOption structure is used to represent the option buffer. This structure is defined to allow easy construction of 4-byte options.

Next, the request parameter for the OTOptionManagement function is defined to reference the option buffer that was just initialized. The request.flags field is initialized to T_CURRENT, specifying that we want to get the current value of the option. The reply parameter for the OTOptionManagement function is then initialized. This is where the function stores the result of the negotiation. Finally, the OTOptionManagement function is called, and its result is checked to see that the option value was read successfully. Any status other than T_SUCCESS or T_READONLY is stored in the err variable.

The next function called by main is SetFourByteOption, shown in Listing 7-5. This routine sets an option and assigns it to the location referenced by value. The endpoint is assumed to be in synchronous mode. If an error occurs, the function returns a negative result. If the option could not be read, a positive result (T_FAILURE, T_PARTSUCCESS, TREADONLY, or T_NOTSUPPORT) is returned.

Listing 7-5 Setting an option value

```
static OTResult SetFourByteOption(EndpointRef ep,
                                    OTXTILevel level,
                                    OTXTIName  name,
                                    UInt32   value)
{
    OTResult    err;
    TOption     option;
    TOptMgmt    request;
    TOptMgmt    result;

    /* Set up the option buffer to specify the option and value to
            set. */
    option.len      = kOTFourByteOptionSize;
    option.level    = level;
    option.name     = name;
    option.status   = 0;
    option.value[0] = value;

    /* Set up request parameter for OTOptionManagement */
    request.opt.buf     = (UInt8 *) &option;
    request.opt.len     = sizeof(option);
    request.flags       = T_NEGOTIATE;

    /* Set up reply parameter for OTOptionManagement. */
    result.opt.buf       = (UInt8 *) &option;
    result.opt.maxlen    = sizeof(option);


    err = OTOptionManagement(ep, &request, &result);

    if (err == noErr) {
        if (option.status != T_SUCCESS)
            err = option.status;
    }

    return (err);
}
```

The SetFourByteOption function is very similar in structure to the GetFourByteOption function, shown in the previous listing. Once again, we use a TOption structure to represent the option buffer and initialize its fields to specify the option and value we want to set. Next, we initialize the request parameter of the OTOptionManagement function to reference the option buffer we just initialized. The length field is set to the size of the option buffer and the flags field is set to T_NEGOTIATE to specify that we want to set the option value specified in the option buffer.

The reply parameter for the OTOptionManagement function is then set up. This is where the function will store the negotiated value of the option when it returns. Finally, the OTOptionManagement function is invoked and its result is checked to make sure the option was successfully negotiated.

After calling the SetFourByteOption function, main calls the GetFourByteOption function again to check the newly set value. The next three functions, PrintAllOptionsAtLevel, PrintOptionsForConfig, and BuildAndPrintOptions demonstrate various ways of displaying option values for an endpoint. Two of those functions call the function PrintOptionBuffer shown in Listing 7-6. This function calls the Open Transport function OTNextOption to parse through an options buffer and then uses printf statements to display the results.

Listing 7-6 Parsing an options buffer

```
static OSStatus PrintOptionBuffer(const TNetbuf *optionBuffer)
{
    OSStatus         err;
    TOption         *currentOption;

    currentOption    = nil;
    do
    {err = OTNextOption(optionBuffer->buf, optionBuffer->len,
                                                &currentOption);
        if (err == noErr && currentOption != nil)
             printf("Level = $%08x, Name = $%08x, Data Length = %d,
                            Status = $%08x\n",
                            currentOption->level,
                            currentOption->name,
                            currentOption->len - kOTOptionHeaderSize,
                            currentOption->status);
    } while (err == noErr && currentOption != nil);
        return (err);
}
```

The PrintOptionBuffer function displays the level, name, size, and status for each option in the option buffer. The PrintAllOptionsAtLevel function, which main calls next, uses this function to display all the options for an endpoint that are set at a specified level.

Listing 7-7 Obtaining options for a specific level

```
static OSStatus PrintAllOptionsAtLevel(EndpointRef ep, OTXTILevel level)
{
    OSStatus        err;
    TEndpointInfo   epInfo;
    TOptionHeader   requestOption;
    void            *resultOptionBuffer;
    TOptMgmt        request;
    TOptMgmt        result;

    resultOptionBuffer = nil;

    /* Find max size of options of endpoint and allocate buffer */

    err = OTGetEndpointInfo(ep, &epInfo);
    if (err == noErr) {
        resultOptionBuffer = OTAllocMem(epInfo.options);
        if (resultOptionBuffer == nil)
            err = kENOMEMErr;
        }

    /* Call OTOptionManagement to get current option values */
    if (err == noErr) {
        requestOption.len    = kOTOptionHeaderSize;
        requestOption.level  = level;
        requestOption.name   = T_ALLOPT;
        requestOption.status = 0;

        request.opt.buf      = (UInt8 *) &requestOption;
        request.opt.len      = sizeof(requestOption);
        request.flags       = T_CURRENT;

        result.opt.buf      = resultOptionBuffer;
        result.opt.maxlen   = epInfo.options;

        err = OTOptionManagement(ep, &request, &result);
    }
    /* Print options to stdout. */

    if (err == noErr) {
        err = PrintOptionBuffer(&result.opt);
        printf("\n");
    }
    if (resultOptionBuffer != nil)
        OTFreeMem(resultOptionBuffer);
    return (err);
}
```

The function `PrintAllOptionsAtLevel` takes two parameters, an endpoint reference and a level value. The function first calls `OTGetEndpointInfo` to determine the maximum size of options for the endpoint and then allocates a buffer to hold the options after they are read. Next, the `OTOptionManagement` function is called to get the current value (`T_CURRENT`) of all the options (`T_ALLOPT`) set for the endpoint. The option values that are returned are stored in the buffer referenced by the `result` parameter. A pointer to the result parameter is passed to the function `PrintOptionsBuffer` (page 182), which displays option values. After the values are displayed, the memory allocated for the result options buffer is freed.

The next function called by `main` is `PrintOptionsForConfiguration`, shown in Listing 7-8. This function gets all the options associated with a level of a provider, converts those options to a formatted string, and then displays that string. The function demonstrates one common use of the function `OTCreateOptionString`.

Listing 7-8 Using the `OTCreateOptionString` function

```
static OSStatus PrintOptionsForConfiguration(const char *configStr,
                                                        OTXTILevel level)
{
```

```
OSStatus        err;
OSStatus        junk;
EndpointRef     ep;
TEndpointInfo   epInfo;
TOptionHeader   requestOption;
void            *resultOptionBuffer;
TOptMgmt        request;
TOptMgmt        result;
TOption         *resultOption;
char            optionsString[1024];

resultOptionBuffer  = nil;
ep                  = kOTInvalidEndpointRef;

/* Create an endpoint using the specified configuration. */
ep = OTOpenEndpoint(OTCreateConfiguration(configStr), 0, &epInfo,
                                                        &err);

/* Allocate a buffer to store option info */

if (err == noErr) {
    resultOptionBuffer = OTAllocMem(epInfo.options);
    if (resultOptionBuffer == nil)
        err = kENOMEMErr;
    }

/* Get the current value of all options at the specified level. */
if (err == noErr) {
    requestOption.len       = kOTOptionHeaderSize;
    requestOption.level     = level;
    requestOption.name      = T_ALLOPT;
    requestOption.status    = 0;
    request.opt.buf         = (UInt8 *) &requestOption;
    request.opt.len         = sizeof(requestOption);
    request.flags           = T_CURRENT;

    result.opt.buf          = resultOptionBuffer;
    result.opt.maxlen    = epInfo.options;

    err = OTOptionManagement(ep, &request, &result);
}
```

```
    /* Convert the options bufferinto a formatted string, and display */
    if (err == noErr)
    {
        resultOption = (TOption *) result.opt.buf;
        err = OTCreateOptionString(configStr, &resultOption,
                result.opt.buf + result.opt.len, optionsString, 1024);
        if (err == noErr)
            printf("Formatted Options = “%s”\n\n", optionsString);
    }

    /* Clean up. */

    if (resultOptionBuffer != nil)
        OTFreeMem(resultOptionBuffer);
    if (ep != kOTInvalidEndpointRef) {
        junk = OTCloseProvider(ep);
        OTAssert("PrintOptionsForConfiguration: Closing the endpoint
                    failed", junk == noErr);
    }
    return (err);
}
```

The PrintOptionsForConfiguration function takes two parameters, a configuration string that describes a specific endpoint provider and a level for which we are interested in getting option information. The function first opens an endpoint using the configuration information passed in. Note that the address of a buffer to hold endpoint information (&epInfo) is also specified in the call. We need the endpoint information structure in order to determine the maximum size of the options buffer for the endpoint.

Next, the function allocates a buffer to hold option values passed back by the OTOptionManagement function. The value of T_ALLOPT for the name field and T_CURRENT for the flags field specify that we are interested in getting currently set values for all options set for the endpoint. After the buffer is allocated, the OTOptionManagement function is invoked. Finally, the OTCreateOptionString function is invoked; this function converts the option buffer passed back by the OTOptionManagement function into a formatted string, which is then displayed using a printf statement.

The last function called by main, BuildAndPrintOptions, accomplishes the reverse of the PrintOptionsForConfiguration function: it takes a configuration string and a set of formatted options, converts the options to their binary format (that is, an options buffer), and then displays the contents of that buffer. The `BuildAndPrintOptions` function is shown in Listing 7-9.

Listing 7-9 Building an options buffer from a configuration string

```
static OSStatus BuildAndPrintOptions(const char *configStr,
                            const char *optionsString)
{   OSStatus         err;
    OSStatus         junk;
    void             *resultOptionBuffer;
    EndpointRef      ep;
    TEndpointInfo    epInfo;
    TNetbuf          optionsNetbuf;

    resultOptionBuffer = nil;
    ep = kOTInvalidEndpointRef;

    /* Create an endpoint using the specified configuration. */

    ep = OTOpenEndpoint(OTCreateConfiguration(configStr), 0,
                                                &epInfo, &err);

    /* Allocate a buffer of the maximum option buffer size. */
    if (err == noErr) {
        resultOptionBuffer = OTAllocMem(epInfo.options);
        if (resultOptionBuffer == nil)
            err = kENOMEMErr;
    }

    /* Parse formatted optionsString into the binary format */
    if (err == noErr) {
        optionsNetbuf.buf = resultOptionBuffer;
        optionsNetbuf.len = 0;
        optionsNetbuf.maxlen = epInfo.options;
        err = OTCreateOptions(configStr, (char **) &optionsString,
                                                    &optionsNetbuf);
        if (err == noErr) {
            err = PrintOptionBuffer(&optionsNetbuf);
            printf("\n");
        }
    }
    /* Clean up. */
    if (resultOptionBuffer != nil)
        OTFreeMem(resultOptionBuffer);
    if (ep != kOTInvalidEndpointRef) {
        junk = OTCloseProvider(ep);
        OTAssert("BuildAndPrintOptions: Closing the endpoint failed",
                                                          junk == noErr);
    }
    return (err);
}
```

The function `BuildAndPrintOptions` creates an endpoint using the specified configuration. When opening the endpoint, the address of an endpoint information structure (`&epInfo`) is passed in; the endpoint provider fills in this structure with information about the endpoint, including its maximum option buffer size. The function then allocates a buffer that is large enough to contain option information for the endpoint.

Next the function `BuildAndPrintOptions` calls the `OTCreateOptions` function to parse the formatted `optionsString` into the binary format (`optionsNetbuf`). Then the function calls `PrintOptionBuffer` (page 182) to display the contents of the options buffer. Finally, the function frees memory allocated for the options buffer and returns.

Ports

Contents

This chapter discusses the concept of ports in Open Transport and introduces the Open Transport functions that provide information about the ports available on your computer.

You need to read this chapter if your application has the ability to use multiple ports, and you need to be able to obtain port information.

This chapter discusses how your application can

- browse available ports and get specific port information
- register as an Open Transport client
- handle yield port requests

This chapter begins by introducing basic concepts about ports, then gives the details of how to find a specific port and extract information about it, and explains how to register your application as an Open Transport client. For complete information about the data types and functions used to work with ports, see "Ports Reference" (page 589).

About Ports

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. When you initialize Open Transport, it scans the local machine for all available ports and creates a data structure called a **port registry** in which it stores information about these ports.

Typically, your application uses whichever port is defined in the appropriate control panel (for example, AppleTalk or TCP/IP). 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.

Identifying Ports

Open Transport provides a standard naming scheme for describing the ports available to a computer. There are two ways to identify each port uniquely: its port name and its port reference.

Port Name

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 the OTCreateConfiguration function 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.

You can identify a port using only the device name, in which case Open Transport uses the first device of that type that has been registered and is available. For most devices this means the built-in device, if one exists. Otherwise, this refers to the first slotted device that has been registered.

Port Reference

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 Port Reference"(page 595).

Multiport Identifiers

The **multiport identifier** is a port function parameter that distinguishes between multiple ports when a single slot supports more than one port of the same device type. This parameter, called other, is part of the port reference structure, which is described in "The Port Reference" (page 595).

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. An example of a multifunction card is 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. A multi-use device is registered with all attributes identical except for the device type.

Pseudodevices

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 accommodate 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.

Port Structures

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 Port Structure" (page 592).

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 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 Name Registery, which is a register of hardware and software configuration information for Power Macintosh computers that is maintained by Mac OS. For more information about the 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 these flags.

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 "ItlkB" default.

Using Ports

This section describes how to obtain port information, and how to register as an Open Transport client.

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 on your computer. Open Transport registers all ports on 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 Port Structure” (page 592).

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 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 can 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” (page 595).

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 608).

If you want to extract information from a port reference, you have to use specific Open Transport functions: OTGetDeviceTypeFromPortRef, OTGetBusTypeFromPortRef, OTGetPortIConFromPortRef, OTGetUserPortNameFromPortRef, and OTGetSlotFromPortRef.

Note

Listing 8-1 shows the user-defined function OTFindSerialPorts. This function uses the OTGetIndexedPort function to find all valid ports. For each port, it gets and examines the device type and, if it’s a serial port and not an alias, it calls the user-defined PrintSerialPort function to output information about the port(Note that you don't want to include aliases to the serial ports in the list, otherwise a standard machine will have 3 serial ports, "serialA", "serialB" and "serial" The PrintSerialPort function uses the OTGetUserPortNameFromPortRef function to print the user name for each port.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. ◆

Listing 8-1 Finding all serial ports

```
static OSStatus OTFindSerialPorts(void)
{
    OSStatus err;
    Boolean portValid;
    SInt32 portIndex;
    OTPortRecord portRecord;
    UInt16 deviceType;
/* Start portIndex at 0 and call OTGetIndexedPort until */
/* there are no more ports. */
    portIndex = 0;
    err = kOTNoError;
    do {
        portValid = OTGetIndexedPort(&portRecord, portIndex);

    /* Get the deviceType; and, if it's a serial port */
    /* and not an alias, call PrintSerialPort */
        if (portValid) {
            deviceType = OTGetDeviceTypeFromPortRef(portRecord.fRef);
            if (deviceType == kOTSerialDevice &&
                        (portRecord.fInfoFlags & kOTPortIsAlias) == 0) {
                err = PrintSerialPortInfo(&portRecord);
            }
        }
        portIndex += 1;
    } while ( portValid && err == kOTNoError);
    return err;
}
static OSStatus PrintSerialPortInfo(const OTPortRecord *portRecord)
{
    Str255 userVisibleName;

/* You must be running PPC codeto call OTGetUserPortNameFromPortRef */
/* on a PPC machine. */

    OTGetUserPortNameFromPortRef(portRecord->fRef, userVisibleName);

    printf("Found a serial port with port reference $%08lx:\n",
                                            portRecord->fRef);
    printf("  User visible name is                       “%#s”.\n",
                                                userVisibleName);
    printf("  String to pass to OTCreateConfiguration is “%s”.\n",
                                            portRecord->fPortName);
    printf("  Name of provider module is                 “%s”.\n",
                                            portRecord->fModuleName);
    printf("\n");
    return kOTNoError;
}
```

Requesting a Port to Yield Ownership

There may be times when you need to use a particular port (normally, a serial port or modem) that is owned by another application. You can use the `OTYieldPortRequest` function to request the owner of a port to yield the use of the port to you. (Check the `kOTPortCanYield` bit in the port record's flags field to determine whether the port supports yielding.) 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, compliance is assured.

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 a short period of time (typically 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 request.

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.

If you use the OTRegisterAsClient function to register a notifier for client events, you would receive the events such as kOTPortDisabled, kOTPortEnabled, kOTPortOffline, kOTPortOnline, kOTClosePortRequest, kOTYieldPortRequest, and kOTNewPortRegistered to keep you informed. For information about port events, see “Port-Related Events” (page 590).

The OTRegisterAsClient function is optional. If you do not want to receive these events, you do not have to call this function.

Calling the CloseOpenTransport function automatically unregisters you as a client. If you want to unregister prior to calling CloseOpenTransport, you can call the OTUnregisterAsClient function.

Note
Client notifiers are distinct from a provider notifier. Provider notifiers tell you about events related to that specific provider. Client notifiers are sent events about the Open Transport system as a whole. ◆

CHAPTER 9

Utilities

Contents

This chapter describes utility functions that you can use to implement time stamps, manipulate lists and strings, manipulate memory, and perform atomic operations.

Open Transport utility functions are described in detail in "Utilities Reference" (page 621).

About Utility Functions

You can use Open Transport utility functions to measure time, manipulate memory, strings, and lists, and to perform atomic operations. These functions have proved useful in the implementation of Open Transport and they have been exported for your use and convenience. In some instances, they can prove more accurate or faster than equivalent functions provided by the operating system.

This section summarizes the utility functions that are available to applications using Open Transport and explains what relative advantages they offer over their operating system equivalents, if any.

- Timing functions

  You can use these functions to obtain the current time, to measure elapsed time, to find the sum or difference between two timestamp values, and to convert timestamp values into microseconds or milliseconds.

  These functions are faster and more accurate than their Time Manager equivalents; they are also more consistent with Open Transport internal timing measurements.

- Memory manipulation functions

  You can use these functions to move memory, to compare two regions of memory, and to initialize memory ranges to set values.

  For 68000 code, these functions are dispatched more quickly than their equivalent Toolbox functions. For PowerPC platforms, performance is roughly equivalent to Toolbox functions. Note that hese functions should be used for cached memory only. Using them for uncached memory will not return an error, but will degrade performance.

- Manipulating lists

  You can use these functions to manipulate entries in FIFO (first-in first-out) and LIFO (last-in first-out) lists.

- Manipulating strings

  You can use these functions to obtain the length of a string, to copy a string, to concatenate strings, and to determine whether two strings are equal. These functions are provided for those writing Open Transport modules and drivers (because these cannot be linked with standard C libraries) but you can also call these functions from application code.

- Using atomic operations

  You can use these functions to test and clear bits, to compare and swap variously-sized values, and to add values. In addition, two atomic operations are provided that allow you to set and clear locks. These are described in "Locking Functions" (page 667).

  The equivalent of these functions are already available on some machines as part of the Driver Services Library. The advantage of using Open Transport atomic operations is that they are available on all platforms and don't depend on the avilability of the driver services library. For 68000 code, Open Transport atomic operations are in-lined and are, therefore, very fast.

Using List Management Functions

The use of most of the utility functions introduced in this chapter and described in "Utilities Reference" (page 621) is fairly straightforward. As mentioned in the previous section, using these utilities can often result in better performance. However, the group of functions used to manipulate linked lists merits additional comment. This section focuses on the use of these functions by way of a sample program, "ListMania," which is designed to illustrate the use of the Open Transport linked-list routines in a simple producer-consumer environment. The code includes two key routines, `ProduceWidgets` and `ConsumeWidgets`.

The ListMania program uses Open Transport LIFO list routines throughout. These routines are atomic with respect to interrupts and threads, and they are faster than the standard Mac OS equivalent functions (`Enqueue` and `Dequeue`). All code included in this sample program is running at system task time;

however, all the list management in the critical portions of the code is perfectly safe to run at interrupt time. This program also demonstrates another advantage of using Open Transport list management functions: they make it easy for you to keep elements on multiple lists simultaneously. For example, in the ListMania program any given widget is always on the `gAllWidgetList` (linked through the `fAllWidgets` field) and on either the `gPendingWidgetList` or the `gFreeWidgetList` (linked through the fNext field).

Before looking at the program itself, we'll briefly discuss the data structures used by the program. The objects being produced and consumed are widgets, as defined by the `Widget` data type:

```
struct Widget {
    OTLink      fNext;
    OTLink      fAllWidgets;
    UInt32      fSequenceNumber;
    OTTimeStamp fCreationTime;
    OTTimeStamp fLastProducedTime;
};
typedef struct Widget Widget, *WidgetPtr;
```

The first two fields are link fields: The `fNext` field is used to link all the elements on either the pending or free widget list; the `fAllWidgets` field is used to link all the widgets in one long list, regardless of their status. The `fSequenceNumber` field is a unique monotonically increasing sequence number for each widget that is created. The `fCreationTime` field specifies the time when a widget is created, and the `fLastProducedTime` field specifies the time when a widget was last produced. The program also uses three LIFO lists: `gAllWidgetList` (which contains all widgets), `gPendingWidgetList`, and `gFreeWidgetList`.

Listing 9-1 shows the the global variable declarations for the ListMania program.

Listing 9-1 ListMania: global declarations

```
struct Widget {
    OTLink      fNext;
    OTLink      fAllWidgets;
    UInt32      fSequenceNumber;
    OTTimeStamp fCreationTime;
    OTTimeStamp fLastProducedTime;
};
typedef struct Widget Widget, *WidgetPtr;

static OTLIFO gAllWidgetList;
static OTLIFO gPendingWidgetList;
static OTLIFO gFreeWidgetList;

static UInt32 gLastWidgetSequenceNumber;
```

The function `InitWidgetLists`, shown in Listing 9-2 initializes all of the widget lists to empty.

Listing 9-2 The InitWidgetLists function

```
static void InitWidgetLists(void)
    /* Initializes all of the widget lists to empty.*/
{
    gAllWidgetList.fHead = nil;
    gPendingWidgetList.fHead = nil;
    gFreeWidgetList.fHead = nil;
    gLastWidgetSequenceNumber = 0;
}
```

The function shown in Listing 9-3creates a widget.

Listing 9-3 The CreateWidget function

```
static WidgetPtr CreateWidget(void)
{
    WidgetPtr result;
/* Allocate the memory for the widget. */
    result = OTAllocMem(sizeof(Widget));
    OTAssert("CreateWidget: Could not create widget", result != nil);

    OTMemzero(result, sizeof(Widget));
    result->fSequenceNumber = OTAtomicAdd32(1, (long *)
                                    &gLastWidgetSequenceNumber);
    OTGetTimeStamp(&result->fCreationTime);

/* Add the widget to the list of all the widgets in the system.*/
    OTLIFOEnqueue(&gAllWidgetList, (OTLink *) &result->fAllWidgets);
    return (result);
}
```

The CreateWidget function allocates memory for a widget and then fills out the information fields for the widget structure: fSequenceNumber and fCreationTime. Note the use of the utility function OTAtomicAdd32 (page 664) to increment the variable gLastWidgetSequenceNumber atomically. This guarantees that the sequence number is unique, even it this routine is re-entered. After creating the widget, the function adds it to the list of all the widgets in the system, gAllWidgetList, using the Open Transport list routine OTLIFOEnqueue (page 641), and then it returns a pointer to the newly created widget.

The ProduceWidgets function (shown in Listing 9-4) either calls CreateWidgets if there are no free widgets or obtains a free widget from the free widget list and then adds the widget to the pending widget list.

Listing 9-4 The ProduceWidgets function

```
static void ProduceWidgets(UInt32 howMany)
{
    UInt32       i;
    OTLink      *freeLink;
    WidgetPtr    thisWidget;

    for (i = 0; i < howMany; i++) {
        freeLink = OTLIFODequeue(&gFreeWidgetList);
        if ( freeLink != nil ) {
            thisWidget = OTGetLinkObject(freeLink, Widget, fNext);
        } else {
            thisWidget = CreateWidget();
        }
        OTGetTimeStamp(&thisWidget->fLastProducedTime);

        OTLIFOEnqueue(&gPendingWidgetList, (OTLink *)
                                                &thisWidget->fNext);
    }
}
```

The for loop used in the function ProduceWidgets takes a free element from the front of the free widget list using the OTLIFODequeue function (page 642). If the function returns nil, there is no free element and a new widget needs to be created by calling the CreateWidget function (page 206). If the free widget list does contain a free element, the OTGetLinkObject macro (page 623) is used to derive the widget from freeLink. After this, the widget is no longer on the free widget list and we can now produce the widget by calling the utility function OTGetTimeStamp (page 635) to set the fLastProducedTime field. Once the widget is produced, it is added to the list of pending widgets using the OTLIFOEnqueue function.

The function ConsumeWidgets, shown in Listing 9-5 first calls the OTLIFOStealList function (page 643) to remove all of the widgets on the pending list and then calls the OTReverseList function(page 644) so that the widgets can be consumed in the same order they were produced. While there are still widgets left on the pending list, the function then calls the PrintWidget function, shown in Listing 9-6, and then adds the most recently consumed widget to the free list by calling the OTLIFOEnque function.

Listing 9-5 The ConsumeWidgets function

```
static void ConsumeWidgets(void)
{
    OTLink   *listToConsume;
    WidgetPtr thisWidget;

/* Remove widgets from pending list; put them in list to consume */
    listToConsume = OTLIFOStealList(&gPendingWidgetList);
    listToConsume = OTReverseList(listToConsume);

    while ( listToConsume != nil ) {
/* Given the link element, derive the actual widget object.*/
        thisWidget = OTGetLinkObject(listToConsume, Widget, fNext);

/* Consume the widget by printing to stdout */
        PrintWidget(thisWidget);
        printf("\n");

/* Get next list element... */
        listToConsume = listToConsume->fNext;
        /* add the most recently consumed widget to free list */
        OTLIFOEnqueue(&gFreeWidgetList, (OTLink *) &thisWidget->fNext);
    }
}
```

It's important to note the order of the two operations used to consume (print) the widget and to add the most recently consumed widget to the free list. This is because the field `thisWidget->fNext` occupies the same memory location as the field `listToConsume->fNext`, so we can't change `thisWidget->fNext` by enqueuing it until we have extracted the linkage information from it.

Listing 9-6 The PrintWidget function

```
static void PrintWidget(WidgetPtr thisWidget)
{
    printf("  %03d, Created @ %08x%08x, Produced @ %08x%08x",
        thisWidget->fSequenceNumber,
        thisWidget->fCreationTime.hi,
        thisWidget->fCreationTime.lo,
        thisWidget->fLastProducedTime.hi,
        thisWidget->fLastProducedTime.lo
        );
}
```

The `DumpAllWidgetLists` function, shown in Listing 9-7 dumps all of the widgets on all of the lists. Because the widgets are linked in different ways on the three lists, the `DumpAllWidgetLists` function must call an additional function, `DumpWidgetList`, to dump the widgets that are linked using the `fNext` field—that is, the widgets in the pending and free lists.

Listing 9-7 The DumpAllWidgetLists

```
static void DumpAllWidgetLists(void)
{
    OTLink      *link;
    WidgetPtr    thisWidget;

    printf("gPendingWidgetList\n");
    DumpWidgetList(&gPendingWidgetList);

    printf("gFreeWidgetList\n");
    DumpWidgetList(&gFreeWidgetList);

    printf("gAllWidgetList\n");
    link = gAllWidgetList.fHead;
    while ( link != nil ) {
        thisWidget = OTGetLinkObject(link, Widget, fAllWidgets);
        PrintWidget(thisWidget);
        printf("\n");
        link = link->fNext;
    }
}
```

The `DumpWidgetList` function is shown in Listing 9-8.

Listing 9-8 The DumpWidgetList function

```
static void DumpWidgetList(OTLIFO *list)
    /* Dump a widget list that is linked using the fNext field. */
    /* This is appropriate for the pending and free lists of widgets. */
{
    OTLink *link;
    WidgetPtr thisWidget;

    link = list->fHead;
    while ( link != nil ) {
        thisWidget = OTGetLinkObject(link, Widget, fNext);
        PrintWidget(thisWidget);
        printf("\n");
        link = link->fNext;
    }
}
```

CHAPTER 10

Advanced Topics

Contents

This chapter examines several topics that might be of interest once you are comfortable with the basic use of endpoints. These include

- acknowledging sends
- sending noncontiguous data
- ccessing data from Open Transport's internal receive buffers
- ending and receiving data without stripping header information

Before you read this chapter, you should be familiar with the material covered in the chapters "Providers"(page 61) and "Endpoints"(page 83).

The chapter "Advanced Topics Reference" (page 673) describes the data structures and functions introduced in this chapter.

Acknowledging Sends

By default, providers do not acknowledge sends. This means that when you send 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 no longer uses the buffer you have allocated for the data. As soon as the function completes, 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 you want the endpoint provider to 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. The advantage of acknowledging sends is that it improves performance at the cost of some added complexity in your code.

Sometimes, due to flow control, 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.

▲ **WARNING**
If you want Open Transport to acknowledge sends, you must make sure that there are no outstanding sends when you close Open Transport; otherwise, you crash. ▲

Because of the complexity of handling flow control, Open Transport performance suffers when the acknowledge sends option is used with noncontiguous data, such as when you pass an `OTData` structure to the `OTSnd` function. Therefore, it is best to avoid this option with non-contiguous data, especially if the last element is a large element.

Only endpoint provider functions are affected by your calling the `OTAckSends` and `OTDontAckSends` functions.

IMPORTANT
If the endpoint acknowledges sends and there are outstanding buffers still in use, you must flush the buffers before closing the endpoint provider. To flush the stream, call the function `OTIOCtl` as follows:

```
OTIOCtl (MyEptref, FLUSHRW, 0);
```

Then, wait until you receive all of the `T_MEMORYRELEASED` events. ▲

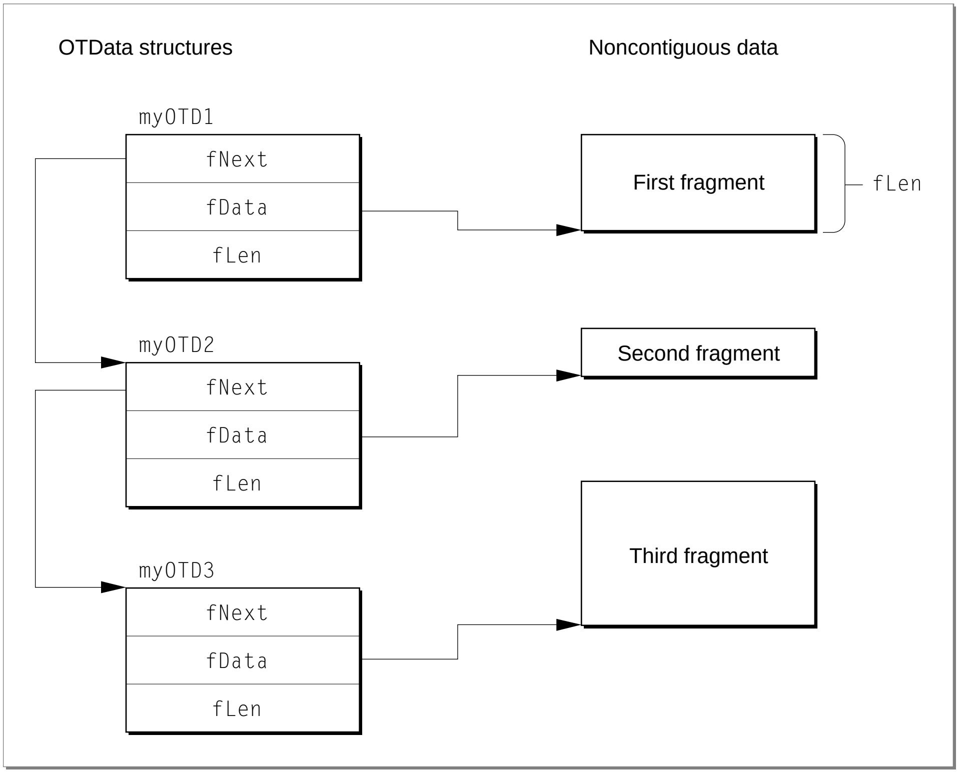
Sending Noncontiguous Data

When sending data, you specify the location and size of the buffer containing the data to be sent (for all send functions except `OTSnd`) using the `len` and `buf` fields of a `TNetBuf` type structure. Open Transport also allows you to send noncontiguous data— data stored in several locations, by using the `OTData` structure to describe that data and passing it as the data buffer. You can send noncontiguous data using the functions `OTSnd`, `OTSndUData`, `OTSndURequest`, `OTSndUReply`, `OTSndRequest`, and `OTSndReply`.

Note
The `OTData` structure and its use in describing noncontiguous data is an Apple extension to the XTI API. ◆

Figure 10-1 shows how you use OTData structures to describe noncontiguous data.

Figure 10-1 Describing 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 last OTData structure to NULL.

When sending noncontiguous data (using the functions OTSnd, OTSndUData, OTSndURequest, OTSndUReply, OTSndRequest, and OTSndReply), the buf field of the TNetBuf structure (or the buf parameter to the function) must point to an OTData structure that describes the first data fragment. You must also set the len field of the TNetBuf structure (or the nbytes parameter to the function) to kNetbufDataIsOTData.

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.

Normally, when you call one of the receive functions to get data, you pass the address of a buffer you have allocated, and Open Transport copies data from its own internal buffers to the buffer you specify. Doing a no-copy receive means that Open Transport does not copy data from its buffers into yours, but instead allows you to access its internal buffers directly. 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 in 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 use this buffer correctly, you must

- not write to it; if you do, you can crash the system
- release it quickly
- only release it once; don't release it multiple times

You need to release the no-copy receive buffer (with the OTReleaseBuffer function) as soon as you are finished using it so that you are not tying up system resources required elsewhere. One consequence of holding on to a buffer too long is that your link layer driver starts allocating more buffers as it receives more data and, if it isn't well designed, it may run out of space and lose packets.

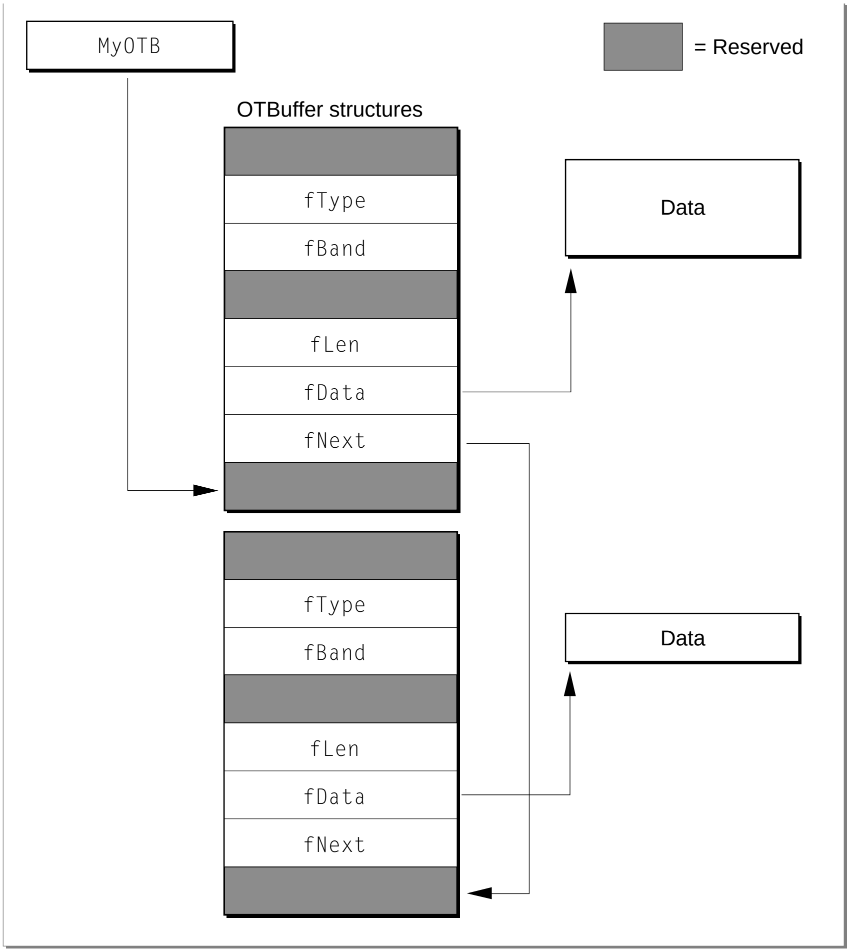
In many cases, for performance reasons, drivers pass their actual DMA buffers when they return data. If this is the case, when you do a no-copy receive, you are getting the actual DMA buffers from the driver. If you hold on to the buffer for too long, you may begin to starve the driver for DMA buffers, 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.

▲ **WARNING**

On PPC systems no-copy receives are only supported for PPC Open Transport clients. Emulated 68000 clients may not use no-copy receives on PPC systems. ▲

Figure 10-2 shows the structure of Open Transport's internal receive buffers. You will be accessing data referenced in these buffers when you do a no-copy receive. To do this, you allocate a variable that holds the address of the first `OTBuffer` structure and then access additional buffers using the `fNext` field of each buffer.

Figure 10-2 OTBuffer structures



In Figure 10-2 the variable `MyOTB` holds the address of the first `OTBuffer` structure. The unused fields of the structure are shaded. The `fData` field of the structure points to the first data packet; the `fLen` field specifies the length of the data packet, and the `fNext` field holds the address of the second `OTBuffer` structure, which provides the location and size of the second data packet.

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.

In order to do a no-copy receive, you must

1. Allocate a local variable into which Open Transport will store the address of the first OTBuffer structure. The OTBufferInfo type is especially useful for this local variable.
2. Pass the constant kOTNetbufDataIsOTBufferStar for the nbytes parameter of the OTRcv function or the udata.maxlen field used with other receive functions to indicate that you are doing a no-copy receive.
3. Use the utility function OTBufferDataSize to determine the size of the no-copy receive buffer.
4. Use the utility function OTReadBuffer to read bytes from the data buffers. The fOffset field of the OTBufferInfo structure specifies how much of the buffer has been read.
5. Use the OTReleaseBuffer function to return the no-copy receive buffer to the system when you are finished copying data from the buffer.

The following two listings show two different methods of doing nocopy receives. Listing 10-1 shows the NoCopyReceiveUsingOTReadBuffer user-defined function. This function reads data from the endpoint (ep) using a no-copy receive. The data is then copied out of the OTBuffer chain using the OTReadBuffer utility function. This method is useful if you need to look at a small chunk of data, which you can copy using OTReadBUffer, to decide what to do with the rest.

Listing 10-1 Doing a no-copy receive: method 1

```
enum {
    kTransferBufferSize = 1024
};

static char gTransferBuffer[kTransferBufferSize];

static OSStatus NoCopyReceiveUsingOTReadBuffer(EndpointRef ep, SInt16
destFileRefNum)
{
```

```
OSStatus err;
OTResult result;
OTBufferInfo bufferInfo;
OTFlags junkFlags;
UInt32 bytesRemaining;
UInt32 bytesThisTime;
SInt32 count;

err = noErr;

/* Initialise the bufferInfo data structure. */

bufferInfo.fOffset = 0;
bufferInfo.fBuffer = nil;

/* Read the data.  Use the constant kOTNetbufDataIsOTBufferStar */
/* to indicate that you want to do a no-copy receive. */

result = OTRcv(ep, &bufferInfo.fBuffer, kOTNetbufDataIsOTBufferStar,
                                            &junkFlags);
if (result >= 0) {

    /* Use OTBufferDataSize to calculate how much data is returned */
    bytesRemaining = OTBufferDataSize(bufferInfo.fBuffer);

    /* Write that data to the file.  We do this in chunks, */
    /* copying each chunk of data out of the OTBuffer chain */
    /* and into our transfer buffer using OTReadBuffer, then */
    /* writing each chunk of data, until there is no
    /* more data left in the buffer chain. */
    while (err == noErr && bytesRemaining > 0) {
        if (bytesRemaining > kTransferBufferSize) {
            bytesThisTime = kTransferBufferSize;
        } else {
            bytesThisTime = bytesRemaining;
        }
        (void) OTReadBuffer(&bufferInfo, gTransferBuffer,
                                            &bytesThisTime);
        count = bytesThisTime;
        err = FSWrite(destFileRefNum, &count, gTransferBuffer);
        bytesRemaining -= bytesThisTime;
        }

        err = noErr;
    } else {
        err = result;
    }

    /* Clean up.  We MUST release the OTBuffer chain back to OT */
    /* so that it can reuse it. OTReleaseBuffer does not tolerate */
    /* the parameter being nil, so we check for that case first. */

    if (bufferInfo.fBuffer != nil) {
        OTReleaseBuffer(bufferInfo.fBuffer);
    }

    return err;
}
```

The method shown in the previous listing is not particularly efficient, but it does demonstrate the use of the OTReadBuffer function. The second method, shown in Listing 10-2 , uses the NoCopyReceiveWalkingBufferChain user-defined function to read data from the endpoint (ep) using a no-copy receive. The code walks through the resulting buffer chain, writing out chunks of data directly to the file from the buffers returned to us by Open Transport.

Listing 10-2 Doing a no-copy receive: method 2

```
static OSStatus NoCopyReceiveWalkingBufferChain(EndpointRef ep,
                                            SInt16 destFileRefNum)
{
    OSStatus err;
    OTResult result;
    OTBufferInfo bufferInfo;
    OTBuffer *thisBuffer;
    OTFlags junkFlags;
    SInt32 count;

    err = noErr;

    /* Initialise the bufferInfo data structure. */
    bufferInfo.fOffset = 0;
    bufferInfo.fBuffer = nil;

    /* Read the data.  Use the constant kOTNetbufDataIsOTBufferStar */
    /* to indicate that you want to do a no-copy receive. */

    result = OTRcv(ep, &bufferInfo.fBuffer, kOTNetbufDataIsOTBufferStar,
                                                &junkFlags);
    if (result >= 0) {

    /* Walk the returned buffer chain, writing out each chunk to file */

        thisBuffer = bufferInfo.fBuffer;
        while (err == noErr && thisBuffer != nil) {

            count = thisBuffer->fLen;
            err = FSWrite(destFileRefNum, &count, thisBuffer->fData);

            thisBuffer = thisBuffer->fNext;
        }
    } else {
        err = result;
    }

    /* Clean up.  We MUST release the OTBuffer chain to Open Transport */
    /* so that it can reuse it. , OTReleaseBuffer does not tolerate */
    /* the parameter being nil, so we check for that case first. */

    if (bufferInfo.fBuffer != nil) {
        OTReleaseBuffer(bufferInfo.fBuffer);
    }

    return err;
}
```

Using Raw Mode

Raw mode refers to the ability of some connectionless providers to pass packet header information, which would normally be stripped at the appropriate protocol level, up to a higher level. For example, if you open a DDP endpoint, you can send and receive data in raw mode in order to determine how many routers a packet had passed through before you receive it. Normally, hop count information is stored in the DDP packet header; by using raw mode you can access this information, which would otherwise be stripped off before you received the packet.

There are two methods for anabling raw mode packet handling. At the link layer (Ethernet and TokenRing), use the `OTOptionManagement` function to enable raw mode packet processing. Above the link layer, the AppleTalk DDP protocol supports raw mode in a different manner. For more information, see the description of the `OTSndUData` function in "Using General Open Transport Functions With DDP" (page 310). Your protocol might support specific options; for example RawIP supports the IP header include option.

Using Option Management to Set Raw Mode

If you want to use raw mode at the link layer level, you should use the option `OPT_SETRAWMODE`. Listing 10-3 shows the user function `DoNegotiateRawModeOption` as an example of how you negotiate raw mode using options.

Listing 10-3 Negotiating raw mode using options

```
#include <OpenTransport.h>
#include <OpenTptLinks.h>

OSStatus DoNegotiateRawModeOption(EndpointRef ep, UInt32 rawModeOption);

/* use the options as defined in the OpenTptLinks.h header
    when setting the rawModeOption parameter. */

OSStatus DoNegotiateRawModeOption(EndpointRef ep, UInt32 rawModeOption)
```

```
{
    UInt8   buf[kOTFourByteOptionSize];
                        /* buffer for fourByte Option size */
    TOption*    opt;    /* option ptr to make items easier to access */
    TOptMgmt    req;
    TOptMgmt    ret;
    OSStatus    err;

    opt = (TOption*)buf;    /* set option ptr to buffer */
    req.opt.buf= buf;
    req.opt.len= sizeof(buf);
    req.flags= T_NEGOTIATE; /* negotiate for rawmode option */

    ret.opt.buf = buf;
    ret.opt.maxlen = kOTFourByteOptionSize;


    opt->level= LNK_TPI;        /* dealing with tpi */
    opt->name= OPT_SETRAWMODE;  /* specify raw mode */
    opt->len= kOTFourByteOptionSize;
    opt->status = 0;
    *(UInt32*)opt->value = rawModeOption;
                /* set the desired option level, true or false */

    err = OTOptionManagement(ep, &req, &ret);

        /* if no error then return the option status value */
    if (err == kOTNoError)
    {
        if (opt->status != T_SUCCESS)
            err = opt->status;
        else
            err = kOTNoError;
    }

    return err;
}
```

The function assumes the endpoint is in synchronous mode. It defines buffers that contain option negotiation information and then calls the OTOptionMangement function.

Testing for Raw Mode Support

To use raw mode you need to determine whether the provider you are using supports it by examining the T_CAN_SUPPORT_MDATA bit of the endpoint information structure for your endpoint. Listing 10-4 shows a function CanDoRawMode that you can call to determine whether your endpoint provider supports sending or receiving raw data. The function calls the OTGetEndpointInfo function and examines the info.flags field to see if the T_CAN_SUPPORT_MDATA bit is set. If it is, the function returns true.

Listing 10-4 Testing for raw data support

```
Boolean CanDoRawMode(EndpointRef ep)
{
    TEndpointInfo   info;
    OSStatus        err;
    Boolean         result;

    err = OTGetEndpointInfo(ep, &info);
    if (err != kOTNoError)
        result = false;
    else if (info.flags & T_CAN_SUPPORT_MDATA)
        result = true;/* this also means that the src addr info is
                            in the info record */
    else
        result = false;

    return result;
}
```

Sending and Receiving in Raw Mode at the Protocol Level

Currently raw mode is supported only for DDP endpoints. To enable or disable raw mode packet processing of data under DDP, you modify the TNetBuf addr field that is sent in the OTSendUdata function. Once raw mode processing is enabled with the OTSndUData call, it stays in effect until you explicitly disable it.

To enable raw mode, you must

1. Specify 0xffffffffUL for the unitdata.addr.len field of the TNetBuf structure containing the address information. Set the opt.len, opt.buf, and addr.buf fields to 0.
2. Place DDP header information in the buffer referenced by the udata.udata.buf field of the TNetBuf structure describing data being sent. The DDP header begins with the hop count byte. With raw mode enabled, the data in the unitdata.udata.buf field must be the complete DDP packet.

Once you have sent a raw mode pocket, the protocol will deliver incoming packets in raw mode as well. When using raw mode on receives, you should

1. Set the opt.len field and the udata.addr.maxlen field to 0. However it is set, Open Transport does not fill this field with address information. Instead it returns the complete DDP packet (including the header) in the data buffer described in step 2.
2. Allocate a buffer (into which the data is stored when the function returns) that is large enough to hold header information as well as the data being received.

Be careful when using raw mode packets because you can no longer tell a full incoming packet from a partial read without remembering that the T_MORE flag was set on the previous read.

To disable raw mode packet processing, send a normal DDP packet with the unitdata.addr.len and the unitdata.addr.buf fields set for an AF_ATALK_DDP or similar structure.

Listing 10-5 shows how you send an echo packet using a DDP endpoint. The sample code includes a call to CanDoMDataMode, a function that looks at the flags associated with creating the endpoint to determine whether the endpoint supports M_DATA mode. It is assumed that the endpoint is bound and that it is in synchronous mode.

Listing 10-5 Testing for raw mode support for a DDP endpoint

```
#include <OpenTransport.h>
#include <OpenTptAppleTalk.h>
#include <Types.h>
#include <Events.h>
#include <stdio.h>

void doOpenTptEcho(EndpointRef ep, UInt16 destNet, UInt8 destNode);
extern Boolean CanDoMDataMode(EndpointRef ep);

enum {
    kddpMaxNormData = 586,
    kddpMaxRawData= 599
};
enum {
    kEchoSocketID= 4
};
enum {
    kEchoRequest= 1,
    kEchoType= 4
};

void doOpenTptEcho(EndpointRef ep, UInt16 destNet, UInt8 destNode)
{
    TBind           boundAddr;
    DDPAddressd     dpAddr, destAddr;
    TUnitData       unitdata;
    OSStatus        err = kOTNoError;
    OTResult        result;
    OTFlags         flags;

    UInt8       buf[kddpMaxRawData];
    UInt8       buf1[64] = "This is a sample string for the first part
                                                 of the buffer";
    Boolean     done = false;
    Boolean     useMDataMode;

    if (!OTIsSynchronous(ep))
    {
        fprintf(stderr, "endpoint must be synchronous for this sample");
        return;
    }

    /* verify that the endpoint is bound first. */
    result = OTGetEndpointState(ep);
    if (result != T_IDLE)
    {
        fprintf(stderr, "endpoint must be bound for this sample");
        return;
    }

        /* check for support of M_DATA mode so that we get the */
        /* header info along with the datagram */
    useMDataMode = CanDoMDataMode(ep);

    if (useMDataMode == true)
    {
/* set up data buffer to send Echo Request as a DDP M_DATA packet */
/* get our protocol address to fill into the M_DATA packet */

        boundAddr.addr.buf = (UInt8*)&ddpAddr;
        boundAddr.addr.maxlen = sizeof(ddpAddr);
        err = OTGetProtAddress(ep, &boundAddr, nil);

        if (err != kOTNoError)
        {
            fprintf(stderr, "error occurred calling OTGetProtAddress
                                    - %ld\n", err);
            return;
        }
        else
        {
            /*  packet length */

            /* clear hopcount, but set the upper 2 bits of the length */
            buf[0] = (UInt8)(kddpMaxRawData >> 8) & 0x0003;
            /* set the lower byte of the length field */
            buf[1] = (UInt8)(kddpMaxRawData & 0x00FF);

            /*  packet checksum */
            buf[2] = 0;// no checksum
            buf[3] = 0;// no checksum

            /*  dest network */

            buf[4] = (UInt8)(destNet >> 8);
            buf[5] = (UInt8)(destNet & 0x00FF);


            /* src network */

            buf[6] = (UInt8)(ddpAddr.fNetwork >> 8);
            buf[7] = (UInt8)(ddpAddr.fNetwork & 0x00FF);

            buf[8] = (UInt8)destNode;       /* dest node */
            buf[9] = (UInt8)ddpAddr.fNodeID;    /* src node */

            buf[10] = kEchoSocketID;/* set dest socket to echo socket */
            buf[11] = (UInt8)ddpAddr.fSocket;/* src socket */

            buf[12] = kEchoType;/* set packet type to echo packet */
            buf[13] = kEchoRequest; /* packet is echo request packet */
            BlockMove((Ptr)&buf1, (Ptr)&buf[14], sizeof(buf1));

                /* set up the unitdata structure */
            unitdata.udata.buf = (UInt8*)buf;   /* data area */
            unitdata.udata.len = kddpMaxRawData;
            unitdata.addr.buf = nil;    /* address area*/

/* by sending the packet with the addr.len field set to 0xFFFFFFFFUL,*/
/* one enables M_DATA mode with DDP.  Once you send a packet in this */
/* manner, all packet deliveries will also be in M_DATA mode.  This */
/* continues until a packet is sent with the addr.len field set to a */
/* value other than 0xFFFFFFFFUL. */
            unitdata.addr.len = (size_t)0xffffffffUL;
            unitdata.opt.buf = nil;
            unitdata.opt.len = 0;   /* no options being sent */
        }
    }
    else
    {
        /* Set up DDP Address field with the destination address */
        /* for the Echo request */

        destAddr.fAddressType = AF_ATALK_DDP;
        destAddr.fNetwork = destNet;
        destAddr.fNodeID = destNode;
        destAddr.fSocket = kEchoSocketID;
        destAddr.fDDPType = kEchoType;

        /* Set up data buffer for the Echo Request */

        /* indicate packet is an echo request packet */
        buf[0] = kEchoRequest;

            /* fill in the buffer with the string */
        BlockMove((Ptr)&buf1, (Ptr)&buf[1], sizeof(buf1));

        /* set up unitdata fields */
        //
        unitdata.udata.buf = (UInt8*)buf;// data area
        unitdata.udata.len = kddpMaxNormData;
        unitdata.addr.buf = (UInt8*)&destAddr;// address area
        unitdata.addr.len = kDDPAddressLength;
        unitdata.opt.len = 0;   // no options being sent
    }

    /* Send the data */
    err = OTSndUData(ep, &unitdata);

    /* If no error occured sending the data, then process */
    /* expected the Echo Response.*/
    if (err == kOTNoError)
    {
        while (done == false)
        {
            result = OTLook(ep);
            if ( result == T_DATA )
            {
                while (done == false)
                {
        /* Set up the UnitData structure to recieve response packet */
        /* Set up the udata and address area to accomodate either an */
        /* M_DATA response or the typical response where the data */
        /* and addr fields are filled in. */

                    unitdata.udata.buf = (UInt8*)buf;/* data area */
                    unitdata.udata.len = 0;
                    unitdata.udata.maxlen = kddpMaxRawData;
                    unitdata.addr.buf = (UInt8*)&destAddr;/* address area
                    unitdata.addr.maxlen = kDDPAddressLength;
                    unitdata.opt.maxlen = 0; /* no options are expected */

        /* note that we reuse the buffer we used to send the echo */
        /* request packet with.  After the OTSnd completes in */
        /* synchronous mode successfully, the buffer has been released */
        /* for use by the program. */
                     result = OTRcvUData(ep, &unitdata, &flags);

                     if (result == kOTNoDataErr)
                     {
                        done = true;
    /* whenever there is a data indication, it's best to read the data */
    /* until the kOTNoDataErr since this releases memory that OT has */
    /* reserved for the data. In this case, we've consumed all */
    /* available data and are ready to exit this function. */
                     }
                     else if (result < 0)
                     {
                        fprintf(stderr, "unknown error occurred reading
                                            data - %ld\n", result);
                        done = true;
                     }
                     else if (result == kOTNoError)
                     {
                        /* read echo reply successfully */
                        /* continue to read until kOTNoDataErr occurs. */
                        fprintf(stderr, "%ld bytes read.\n",
                                                unitdata.udata.len);
                     }
                }

            }
```

```
            else
            {
            /* an event other than T_DATA occurred */

            }
            /* another way to escape this routine. */
            if (Button())
                done = true;
        }
    }
}
```

CHAPTER 11

TCP/IP Services

Contents

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.

This chapter describes the progamming interface to Open Transport's implementation of TCP/IP, including the use of Open Transport endpoint and mapper functions with TCP/IP and the use of single linked multi-homing (available only for Open Transport version 1.3). 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"(page 5), "Getting Started", "Providers"(page 61), "Endpoints"(page 83), "Mappers"(page 149), and "Option Management"(page 165) in this book. For complete reference information about the structures and functions introduced in this chapter, see "TCP/IP Services Reference" (page 683).

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. You should read "About the TCP/IP Protocol Family" (page 237) for an introduction to the protocol family and for pointers to more information.

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.

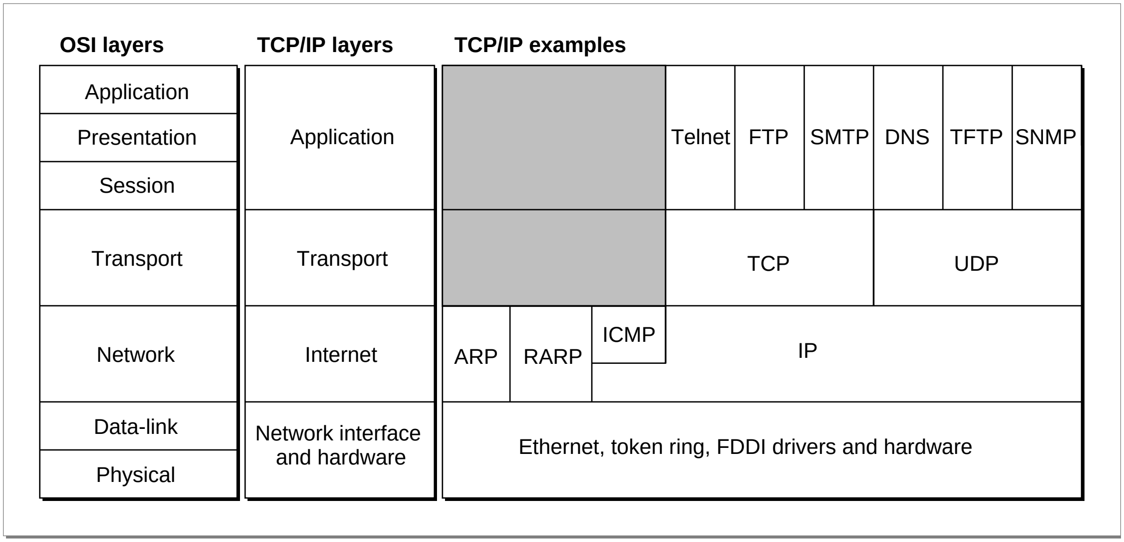
This chapter starts with a brief introduction to the TCP/IP protocol family, followed by an introduction to the TCP/IP services provided by Open Transport. Then "Using General Open Transport Functions With TCP/IP" (page 253) describes TCP/IP-specific information relating to functions described in the chapters "Endpoints" and "Mappers" in this book.

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 performs checksums; 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 11-1 shows the TCP/IP functional layers and examples of TCP/IP protocols that run in each layer. For purposes of comparison, Figure 11-1 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 11-1 TCP/IP protocols and functional layers



As discussed in the chapter “Endpoints”(page 83) 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 11-1 shows how the TCP/IP protocols provided with

Open Transport fit into this matrix. Notice that Open Transport TCP/IP offers no transaction-based protocols.

Open Transport provides an application interface to the IP protocol known as **RawIP**, as shown in Table 11-1. For more information on this interface to the IP protocol, see "Using RawIP" (page 247).

Table 11-1 The Open Transport protocol matrix and TCP/IP protocols

	Connectionless	**Connection-oriented**
Transactionless	RawIP UDP	TCP
Transaction-based	none	none

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 11-1, 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)*. Among other topics, the XTI specification provides detailed descriptions of the sizes and valid settings of the TCP, IP, and UDP options available under Open Transport.

The TCP/IP protocols are defined in a series of documents called Requests for Comments (RFCs). RFCs are available over the Internet at <ttp://nic.ddn.mil>, 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, you can use a file transfer protocol (FTP) client to 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 host 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").

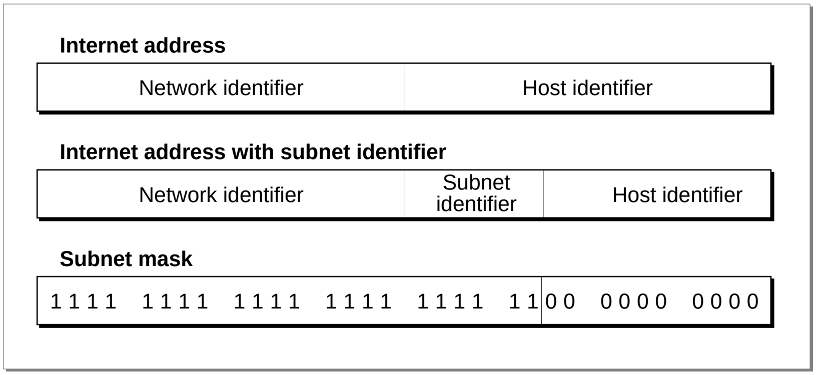
To use the application interface to Open Transport's DNR, you must first open a TCP/IP service provider. Once you have done so, you can

- 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 **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 11-2 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 11-2 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. ◆

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

About the Open Transport DNR

The functions described in "Resolving Internet Addresses" (page 700) and "Getting Information About an Internet Host" (page 705) are implemented by the Open Transport domain name resolver (DNR). The DNR also implements the `OTLookupName` function (page 259) when you create a DNR 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.

The Open Transport DNR implements only the following specific 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.

In addition, the Open Transport DNR provides a generic interface, allowing the user's application to send queries of any type. However, the application is then responsible for parsing the response.

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.

About Single Link Multi-Homing

Open Transport version 1.3 introduces single link multi-homing, a mechanism by which Open Transport can support multiple IP addresses on the same hardware interface. This is useful for users that want to give each of their clients a distinct IP address without requiring separate computers for each address.

In order to use this feature, you must check to see that you are using version 1.3 or later. See "Checking for Availability" (page 245).

Configuring Your System to Use Multiple IP Addresses

To configure your system, you must do the following:

1. Configure the TCP/IP control panel for manual addressing.
2. Create a text file with the name "IP Secondary Addresses".
3. Place the file in the Preferences folder.

Each line of the file contains a secondary IP address to be used by the system, and an optional subnet mask and router address for the secondary IP address. If there is not subnet mask entry, then Open Transport will use a default subnet mask for the IP address class. If there is not router entry, the default router associated with the primary address will be used.

Listing 11-1 shows a sample IP Secondary Addresses file. Each secondary address listed in the file must be prefixed by `ip=`. Each subnet mask entry must be prefixed by `sm=`. Each router address entry must be prefixed by `rt`. Note that the order of the entries is important. The router entry must follow the secondary name entry.

Listing 11-1 Sample IP Secondary Addresses file

```
ip=17.201.22.200    sm=255.255.255.0    rt=17.201.20.1
ip=17.201.22.201                        rt=17.201.20.1
ip=17.201.22.202
```

When Open Transport activates TCP/IP, it obtains the primary address from the TCP/IP Control Panel setting. Open Transport then looks for the IP Secondary Addresses file in the Preferences folder, to determine what other IP addresses the system must support. If it finds duplicate IP address entries in the IP Secondary Addresses file, it ignores them. When Open Transport binds a TCP/IP connection, if there is an address conflict of the primary or any secondary addresses with another host system, Open Transport will raise an Alert with an error message to this problem.

Checking for Availability

To check whether Open Transport version 1.3 is present, use the `Gestalt` function with the `gestaltOpenTptVersions 'otvr'` selector. If the result is greater than or equal to `kOTIPSingleLinkMultihomingVersion`, you can use this feature.

Getting Information About Secondary Addresses

You can use the function `OTGetInterfaceInfo` to return information about the number of secondary addresses that are supported. The `fIPSecondaryCount` field of the structure returned by this function specifies that number. Then, you can call the function `OTInetGetSecondaryAddresses`, passing this number for the `count` parameter, to obtain all the addresses. For more information, see "Single Link Multi-Homing" (page 716).

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" (page 691).

Setting Options When Configuring a TCP/IP Provider

When you open a TCP/IP provider, you must pass a pointer to a configuration string. 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

11-2 provides the constant name and the value to use in the configuration string.

Table 11-2 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"
TCB_MAXSEG	"MaxSeg"
TCP_NOTIFY_THRESHOLD	"NotifyThreshold"

Table 11-2 Configuration strings for TCP/IP options (continued)

Constant name	Configuration string value
TCP_ABORT_THRESHOLD	"AbortThreshold"
TCP_CONN_NOTIFY_THRESHOLD	"ConnNotifyThreshold"
TCP_CONN_ABORT_THRESHOLD	"ConnAbortThreshold"
TCP_KEEPALIVE	"KeepAlive"

The network configuration structure and OTCreateConfiguration function are described in the chapter "Getting Started"(page 31) in this book.

Using RawIP

The Open Transport TCP/IP software modules provide a RawIP interface to the IP protocol. RawIP behaves for the most part identically to UDP, as a connectionless transactionless interface, but there are a few unique differences.

You can receive RawIP datagrams using a RawIP endpoint. You can create a RawIP endpoint by passing kRawIPName to OTCreateConfiguration and passing that configuration to the OTOpenEndpoint or OTOpenEndpointAsync function.

The RawIP interface facilitates 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 datagram header. RawIP endpoints default to receiving ICMP (protocol 1) packets. You can change this by setting the generic XTI option XTI_PROTOTYPE, described in the chapter "Option Management"(page 165) in this book. The option is a longword that is the IP protocol number to be used by the RawIP endpoint.

The data delivered to a RawIP endpoint includes the full IP header, which is 20 bytes long if it includes no IP 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 exercise caution when using Raw IP. ▲

Receiving RawIP Datagrams

Normally, connectionless transactionless endpoints only support binding one endpoint to any given protocol address. RawIP is different in that it allows multiple endpoints to be bound to the same protocol address.

With one important exception, each RawIP endpoint bound to a specific protocol receives a copy of any inbound packets destined for that protocol. For example, if several "ping" programs are using ICMP on the same host, each would receive a copy of all inbound ICMP echo datagrams. The exception is that RawIP endpoints do not receive copies of packets addressed to IP protocols TCP (protocol 6) or UDP (protocol 17). This restriction optimizes the delivery of such packets to their corresponding high-level protocols.

One unusual behavior of RawIP endpoints is that the delivered packets have their `Total Length` field modified. The RawIP module subtracts the length of the IP header from the `Total Length` field. This behavior brings Open Transport's STREAMS RawIP more in line with RawIP under BSD UNIX. Therefore, you should not rely on the value of the `Total Length` field. If you need to know the total length of the packet, use the length as returned in the `TNetBuf` structure returned by the `OTRcvUData` function.

Sending RawIP Datagrams

You can also send RawIP datagrams using a RawIP endpoint. For sending, RawIP endpoints have two modes: a mode in which the RawIP interface generates the header for you and a mode in which you set the header yourself. The header-generated mode is the default, and it is useful if you are only interested in the payload of the RawIP packets you send.

For applications such as ping (ICMP), you can let the RawIP interface generate the headers using the RawIP endpoint default behavior, such as sending ICMP packets (protocol 1). In this case, you can change the protocol and IP options (such as `IP_OPTIONS` and `IP_TTL)` using option management functions, as described in the chapter "Option Management" (page 165) in this book.

Manually Setting the IP Header

At times, however, the level of control provided by the IP level options is not enough. If you need to set a field in the IP header that is not handled by a defined option, you can do this by switching the RawIP endpoint to what is referred to as the header-included mode and setting up the IP header manually.

Internally, the RawIP module maintains a state that determines whether it should add an IP header to any outgoing packets. If the state is false (0), RawIP will automatically generate an IP header for any outgoing packets. If the state is true (1), RawIP expects the data you provide it to contain the IP header. (By default the state is false and RawIP generate headers for you automatically.)

You can change this bit explicitly using option management. Simply set the IP option IP_HDRINCL to a 4-byte integer containing either 0 or 1. The IP options are listed in "IP Options" (page 694).

You can also change this state by changing the IP protocol (using the generic XTI option XTI_PROTOTYPE option) for the endpoint. If you set the IP protocol to IPPROTO_RAW (255) or IPPROTO_IGMP (2), the IP optionIP_HDRINCL state will be set to true. If you change the IP protocol to any other value, the IP option IP_HDRINCL state defaults to false.

IP protocol information sources

The IP Protocol option values are defined in Internet Standard 1 "Assigned Numbers," which can be found at <ftp://ds.internic.net//std/std1.txt>. The fields in the IP header are those defined in Internet Standard 5, which can be found at <ftp://ds.internic.net/std/std5.txt>. ◆

Limitations of the Header-Included Mode

If you use the header-included mode, you need to be aware of some of its limitations. A number of the fields in the IP header are automatically modified by Open Transport, regardless of what values you set them to. These field names include:

- **Version**. This field is forced to a value of 4 to reflect the fact that you're using IP version 4.
- **IHL**. When OT sends a RawIP packet in "header included" mode, it ignores the IHL field value you specify and instead attaches the IP options that were last specified using the IP option IP_OPTIONS. This prevents you from setting your own IP options by placing them in the IP header and setting IHL appropriately.
- **Total Length**. This field is not touched by RawIP, but it must be less than the link MTU for the packet to be sent.
- **Identification**. This field value is set to the next ID number in the Open Transport internal sequence.

Flags. The More Fragments (MF) bit and the reserved bit are cleared. The Do Not Fragment (DF) bit is set on all outgoing IP packets. OT uses the DF bit to implement its dynamic path MTU discovery. Because this behavior is implemented below the IP layer, you cannot change this behavior using the RawIP endpoint

- **Fragment Offset**. This field is completely overwritten by RawIP.
- **Header Checksum** . The field is set to the correct checksum value.

You need to be careful when setting your own IP header. Even though some fields are automatically "corrected" by Open Transport, it is still possible to generate improperly formatted IP packets using a RawIP endpoint, which can result in loss of network data.

Note

Path MTU is described in RFC1191 (ftp:ds.internic.net/rfcl/rfc1191.txt). ◆

Using IP Multicasting

Open Transport TCP/IP provides IP multicasting level 2, as described in RFC 1112. This feature is only relevant for RawIP and UDP endpoints.

To join a multicast group, use the `IP_ADD_MEMBERSHIP` option (page 694), 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 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 throughout 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 option `IP_MULTICAST_LOOP` to turn off loopbacks.

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, and 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 information 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. Each DNS Query Information structure might then contain 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 of pointers to the individual responses.

Avoiding Delay When Rebinding to TCP Connections

When a connection closes, TCP imposes a two-minute timeout on binding before the same port can be bound to again. This prevents stale data from corrupting a new connection. This is in strict compliance with the TCP standard.

You can work around this by using the `IP_REUSEADDR` option with the `OTOptionManagement` function. If you set this option on all of your listening endpoints before you bind, the limitation should disappear. The `IP_REUSEADDR` option allows you to bind multiple connected or closing endpoints to addresses with the same port number.

IMPORTANT

Note that even using the IP_REUSEADDR option, you can only bind a single endpoint in a state less than connected (that is, listening or unbound endpoints) to the same port at a given time. You can, however, bind any number of connected or closing endpoints. ▲

The sample code shown in Listing 11-2 sets an option and assigns it to the location referenced by value. The endpoint is assumed to be in synchronous mode. If an error occurs, the function returns a negative result. If the option could not be read, a positive result (either T_FAILURE, T_PARTSUCCESS, or TREADONLY, or T_NOTSUPPORT) is returned.

Listing 11-2 Setting an option value

```
static OTResult SetFourByteOption(EndpointRef ep,
                                    OTXTILevel level,
                                    OTXTIName  name,
                                    UInt32   value)
{
    OTResult    err;
    TOption     option;
    TOptMgmt    request;
    TOptMgmt    result;

    /* Set up the option buffer to specify the option and value to
            set. */
    option.len      = kOTFourByteOptionSize;
    option.level    = level;
    option.name     = name;
    option.status   = 0;
    option.value[0] = value;

    /* Set up request parameter for OTOptionManagement */
    request.opt.buf     = (UInt8 *) &option;
    request.opt.len     = sizeof(option);
    request.flags       = T_NEGOTIATE;

    /* Set up reply parameter for OTOptionManagement. */
    result.opt.buf       = (UInt8 *) &option;
    result.opt.maxlen    = sizeof(option);


    err = OTOptionManagement(ep, &request, &result);

    if (err == noErr) {
        if (option.status != T_SUCCESS)
            err = option.status;
    }

    return (err);
}
```

In the body of the function, we use a TOption structure to represent the option buffer and initialize its fields to specify the option and value we want to set. Next, we initialize the request parameter of the OTOptionManagement function to reference the option buffer we just initialized. The length field is set to the size of the option buffer and the flags field is set to T_NEGOTIATE to specify that we want to set the option value specified in the option buffer.

You could invoke this function and set the IP_REUSEADDR option as follows:

```
err = SetFourByteOption(ep, INET_IP, IP_REUSEADDR, true);
```

Using General Open Transport Functions With TCP/IP

This section describes special considerations 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 Reference”(page 436) and “Mappers Reference” (page 550)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.

IMPORTANT

The preceding table shows only what values are possible for each protocol. Be sure to to use the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, or `OTGetEndpointInfo` functions to obtain the current values for these parameters. ▲

These fields and the significance of their values are described in more detail in "Endpoints Reference"(page 436).

Using Endpoint Functions With TCP/IP

This section describes protocol-specific information about functions described in the chapter "Endpoints Reference" (page 421). 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. Use this function with the TCP and UDP protocols.

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" (page 685), 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` for the `addr` field 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 bind to an address of `kOTAnyInetAddress`, then the `OTGetProtAddress` function always returns an IP address of 0. In that case, you must use the `OTInetGetInterfaceInfo` function (page 711) to determine the IP address of a running IP interface. However, if you pass in a valid address with a port number of `kOTAnyInetAddress`, the TCP/IP service provider assigns a port for you and the `OTGetProtAddress` function returns the assigned port number and the IP address.

You can use the `OTInetGetInterfaceInfo` function 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 connection requests. Use this function with the TCP 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

You use this function with the TCP and UDP protocols.

If you bind an endpoint to an IP address of `kOTAnyInetAddress` 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 711) 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.

The `rcvcall->addr` field returns a copy of the `TNetbuf` structure you specify in the `sndcall->addr` field. The `discon->reason` field contains a positive error code that indicates why the connection was rejected.

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 the value of the `sndcall->udata.buf` field.

Note that TCP, not the receiving application, confirms the connection.

As mentioned in the X/Open Transport Interface (XTI) specification, because TCP cannot refuse a connection, `t_listen()` and `t_accept()` have a semantic which is slightly different from that for ISO providers."

As a result, an Open Transport TCP server will accept a TCP connection request if the current number of pending connections is less than the queue length (qlen) for the passive endpoint. Basically, what happens is that TCP connects even before you accept a connection.

The client, whether in synchronous or asynchronous mode, will immediately receive notice that the connection has been established. For synchronous endpoints, TCP completes the 3-way connection handshake. For asynchronous endpoints, the OTRcvConnect function must be called to complete the handshake.

This can result in situations like this: You send an OTConnect from a TCP client to a TCP server that passively awaits incoming connections, but even before the server responds with the OTListen and OTAccept calls, the OTConnect call completes with no error. At this point, if you examine the client endpoint's state, you will find that it is in the T_DATAXFER state, which is correct.

OTRcvConnect

The OTRcvConnect function reads the status of a previously issued connection request. You can use this function with TCP.

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 the value of the call->udata.buf field.

On return, the call->addr field points to the 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.

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.maxlen field to 0. TCP ignores the value of the call->udata.buf field.

OTAccept

The OTAccept function accepts an incoming connection request. You can use this function with TCP.

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 the value of 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.

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.

Because it does not support TSDU’s, 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.

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.

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.

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 the value of the discon->udata.buf field.

This function returns a positive error code. To obtain the negative error code, subtract that positive value from -3199.

Using Mapper Functions With TCP/IP

This section describes protocol-specific information about functions described in the chapter “Mappers Reference”(page 550) 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 685), which includes the address type, the port number, and the IP address. 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 `OTInetStringToAddress` function (page 700).

CHAPTER 12

Introduction to AppleTalk

Contents

This chapter provides an overview of the Open Transport implementation of AppleTalk, a communications network system that interconnects 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. Its chief features are that

- it is built into all Mac OS computers
- it provides dynamic addressing, which allows for very easy setup and configuration
- it provides easy resource browsing

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 such as AppleTalk Data Stream Protocol (ADSP) or Printer Access Protocol (PAP) than with a connectionless protocol such as 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 concepts
- 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.

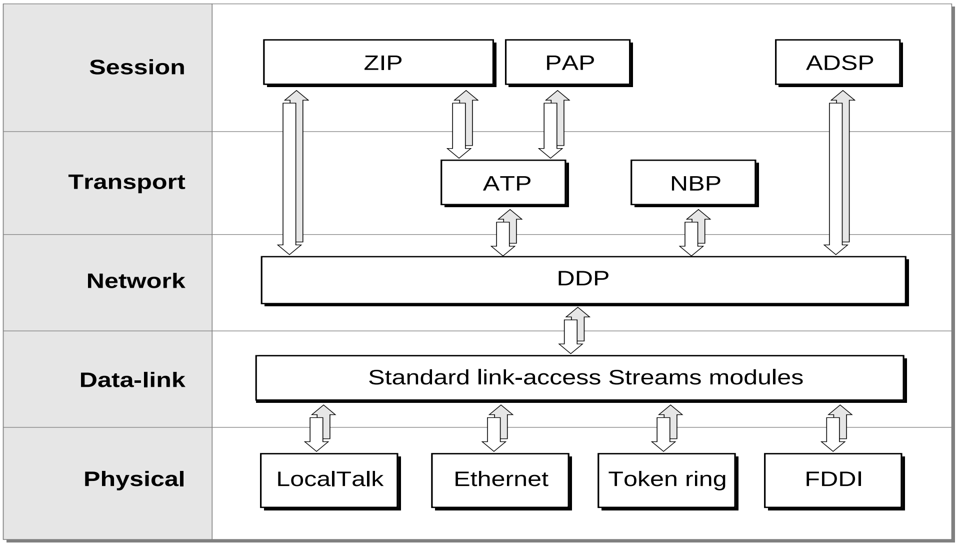
About AppleTalk

Every Mac OS computer includes AppleTalk hardware and software, so if your application needs to communicate with other Mac OS computers, you may want to use an AppleTalk protocol. AppleTalk includes protocols that handle file sharing, 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), 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 12-1 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 12-1 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

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 12-1 summarizes the identifiers that you use for AppleTalk addressing.

Each network is assigned a **network number** so that an AppleTalk 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 the 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

Table 12-1 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.

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
Open Transport allows system administrators to assign static node IDs. ◆

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. the 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 the remote service. This mapping is discussed further in "About AppleTalk Addressing" (page 280) and "About AppleTalk Service Providers" (page 294) 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 multiple 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.

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 Server (ARA), which uses multinodes to make the connected remote client appear on the local network.

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 network (DDP) layer, which means that an application that uses a multinode cannot expect to be supported by 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. An example of such an event 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. (Any applications that rely on the AppleTalk Transition Queue must use AppleTalk backward compatibility to handle them as described in *Inside Macintosh: Networking*.)

In Open Transport AppleTalk, there are five miscellaneous events that you can receive on your AppleTalk 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 computer's zone.
`T_ATALKCONNECTIVITYCHANGEDEVENT`	0x23010054	A 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 "OTIoctl" (page 411) 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 "Configuring and Opening a Provider" (page 34) in this book.

Table 12-2 lists the constants you use to configure the AppleTalk providers. You can use either the constant or the literal string when creating configurations.

Table 12-2 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 12-3 provides the constant name, the value used in the configuration string, and the protocols that use that option.

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);
```

Table 12-3 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

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 "Introduction to Open Transport" (page 5) 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 12-4.

Table 12-4 Open Transport support for AppleTalk endpoint protocols

	Connectionless	**Connection-oriented**
Transactionless	DDP	ADSP PAP
Transaction-based	ATP	

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 various ways of doing this: for example, using the `OTBind` function or opening a mapper provider. 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

See "Mappers" (page 149) for information about how to use Open Transport mapper providers and "AppleTalk Service Providers" (page 293) for information about how to use NBP mapper providers to identify and locate endpoints on a network. The two methods of registering a name are discussed in greater detail in "Registering Your Endpoint's Name" (page 286).

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.

See "AppleTalk Service Providers" (page 293) for information about 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 real-time applications, or 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.

See "Datagram Delivery Protocol (DDP)" (page 303) for information about 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.

See “AppleTalk Data Stream Protocol (ADSP)” (page 313) for information about 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.

See “AppleTalk Transaction Protocol (ATP)” (page 325) for information about 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.

See “Printer Access Protocol (PAP)” (page 335) for information about how to use PAP under Open Transport.

CHAPTER 13

AppleTalk Addressing

Contents

This chapter describes how to specify an AppleTalk address to bind an endpoint, to connect to an AppleTalk service 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 name, a network 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 various ways of doing this, but in either case, Open Transport uses the Name Binding Protocol (NBP) to associate the endpoint's name with its network address. Open Transport provides several utility functions and a specialized data structure, the NBP entity, for more convenient manipulation of NBP names.

This chapter introduces 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. For complete reference information, see "AppleTalk Addressing Reference" (page 721).

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"(page 99) and "Mappers"(page 150) 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, a node's address on an AppleTalk network can change from time to time. The **Name-Binding Protocol (NBP)** provides a mapping of names (like those in the Chooser) to network addresses in such a way that if the node ID changes, you can continue to reliably identify your application. An endpoint's name is its **NBP name,** also sometimes called its *entity name.* You can access information about an endpoint's address through an NBP 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 an address, which can be in one of these formats:

- The **DDP address** supplies the network address of an endpoint.
- The **NBP address** supplies the user-friendly NBP name.
- The **combined DDP-NBP address** combines the endpoint's 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.

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
- unregister an NBP name prior to closing an AppleTalk endpoint

Specifying a DDP Address

The primary address format is the DDP address format, which is the most commonly used. It identifies the network 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 location. Functions that use the NBP address format, for example, have to look up the mapping of the NBP name to its 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, in this case `AF_ATALK_DDP`.
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.

Permissible values for these fields are given in the section "The DDP Address Structure" (page 722). 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)" (page 305) in this book.

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. When you bind an AppleTalk endpoint, you typically specify a network number of 0 and a node ID of 0. This allows the network layer to choose an appropriate address.

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 13-1 shows how you set up the fields of a DDP address and how you set up a `TNetbuf` structure for it.

Listing 13-1 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*) OTAllocMem(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 network 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” (page 287) for more information on how Open Transport translates an NBP name into a network 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, in this case `AF_ATALK_NBP`.
NBP name buffer	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" (page 723).

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 neither a C nor a Pascal string; its length is determined by the `TNetBuf` structure in which it's enclosed. It has the form

name:type@zone

The name field typically identifies the name of the entity on the network; for example, the name of a file server or printer. Another example might be the use of the name in personal file sharing, where the name field is used to register the computer name. Clients can use that name to identify the computer they're logging into.

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 file servers or 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"(page 295) in this book.

You may not use the AppleTalk NBP wildcard characters as part of the NBP name, type, or zone. 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 13-2 shows how you set up the fields of an NBP address. The statements used to set the size of the `len` field 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 13-2 Setting up an NBP address

```
void DoCreateNBPAddress(TNetbuf *theNetBuf, char* nbpName)
{
    NBPAddress *nbpAddress;
    short nbpSize;

    /* Allocate memory for an NBP structure. */
    nbpSize = sizeof(OTAddressType) + OTStrLength(nbpName);
    nbpAddress = (NBPAddress*) OTAllocMem(nbpSize);

    /* Set the TNetbuf to point to it. */
    theNetbuf->len              = OTInitNBPAddress(nbpAddress, nbpName);
    theNetbuf->buf              = (void*)nbpAddress;
}
```

Specifying a Combined DDP-NBP Address

You use the combined DDP-NBP address format (AF_ATALK_DDPNBP) 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" (page 724) 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 (AF_ATALK_MNODE) 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" (page 281) and the section "The Multinode Address Structure" (page 726) 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 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 socket you want to bind the endpoint to. The `OTBind` function is discussed in the chapter "Endpoints" (page 99) in this book.

The other way to register an endpoint's name involves several additional steps. You have to first bind your endpoint to a DDP address, 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 sometimes more convenient than the alternative `OTDeleteName` function. The `OTRegisterName`, `OTDeleteName`, and `OTDeleteNameByID` functions are discussed in the chapter "Endpoints" in this book. Table 13-1 provides a summary of the Open Transport name-registration functions.

Table 13-1 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 with OTRegisterName.
OTDeleteNameByID	Mapper	Given ints identifier , removes a name that was previously registered with OTRegisterName.

Looking Up Names and Addresses

To communicate with an endpoint, Open Transport needs its DDP 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 DDP address that it can use to locate the endpoint you want. Table 13-2 provides a summary of the Open Transport functions that create or return endpoint name and address information.

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 DDP 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.

Table 13-2 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 DDP address.
`OTResolveAddress`	Endpoint	Obtains the DDP address that corresponds to the specified NBP name.
`OTLookUpName`	Mapper	Obtains the DDP address for the specified name or a list of addresses for the specified NBP 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.

When you call the `OTLookUpName` function to obtain the DDP 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 13-3 shows the wildcard operators that you can use to specify a name pattern for a name specified as a partial name.

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.

Table 13-3 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.

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, by using `OTSetNBPName`, `OTSetNBPType`, and `OTSetNBPZone` functions, or you can load an NBP entity with an entire NBP address at one time with 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 with the `OTGetNBPEntityLengthAsAddress` function. You can then extract each individual NBP name part one at a time by using the `OTExtractNBPName`, `OTExtractNBPType`, and `OTExtractNBPZone` functions, or you can copy the entire NBP entity into an NBP address structure wirhthe `OTSetAddressFromNBPEntity` function.

When you no longer need a specific NBP name to be associated with an endpoint, you can use the `OTDeleteName` function `or the OTDeleteNameByID` function to unregister the name.

CHAPTER 14

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 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 complete information about the functions and data structures introduced in this chapter, see "AppleTalk Service Provider Reference" (page 745).

For an overview of the AppleTalk service provider and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk"(page 264). Zones are part of the NBP name used in the NBP address format; for more information on this format, read the chapter "AppleTalk Addressing" (page 280). For a detailed description of the ZIP specification, see *Inside AppleTalk*, second edition.

About AppleTalk Service Providers

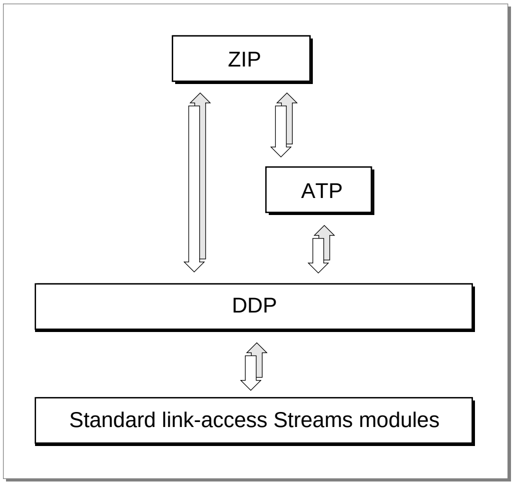
The AppleTalk service provider gives applications access to information and services that are specific to the AppleTalk protocol family. For example, you can obtain zone names and information about the AppleTalk environment for a given machine. The portion of ZIP implemented by AppleTalk service provider 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. An AppleTalk router implements the full set of ZIP functions, 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 14-1 shows how, in providing access to the Zone Information Protocol (ZIP), AppleTalk service providers encompass underlying delivery protocols and link-access STREAMS modules. Because some AppleTalk service provider functions use AppleTalk Transaction Protocol (ATP) packets and DDP, an AppleTalk service provider is considered a client of both ATP and the Datagram Delivery Protocol (DDP).

Figure 14-1 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, the AppleTalk control panel calls 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"(page 150). For example, you can use an AppleTalk service provider to look up all the zones on the network, then use the mapper function OTLookUpName to look up the names in each zone.

Creating 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 open the AppleTalk service provider asynchronously, you need to specify a notifier function that the provider can use to send you completion events and other function-specific information. This notifier API is the same as the one you need to use for asynchronous endpoints.

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” (page 61).

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.

Note that the 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.

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.

For more information about using zones in NBP names and addresses, see the chapters "Introduction to AppleTalk"(page 263) and "AppleTalk Addressing"(page 279) .

Getting the Name of an 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 14-1 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 14-1 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;
        result = OTATalkGetMyZone(svcRef, &zoneNetbuf);
        zoneName[zoneNetBuf.len] = '\0';
        OTCloseProvider(svcRef);
    }
    return result;
}
```

Getting a List of Zone Names

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 the name of the zone, 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" (page 296) for more information about buffer sizes.

Getting Information About the 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 on 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 extended or nonextended, this function provides your application's network address and the address of a local router. If your application's network is extended, this function also sets a flag indicating that it's an extended network and provides 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.

CHAPTER 15

Datagram Delivery Protocol (DDP)

Contents

This chapter describes the programming interface to Open Transport's implementation of 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's faster than the higher-level protocols that add these services. Applications 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 your application requires a reliable service, and you do not want to implement it yourself, you should consider using a higher-level 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 General Open Transport Functions With DDP"(page 310) 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 "Endpoint" (page 83) in this book.

For complete reference information about DDP options, see "DDP Reference" (page 756). For an overview of DDP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" (page 263), 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"(page 279). For a complete explanation of the DDP specification, see *Inside AppleTalk*, second edition.

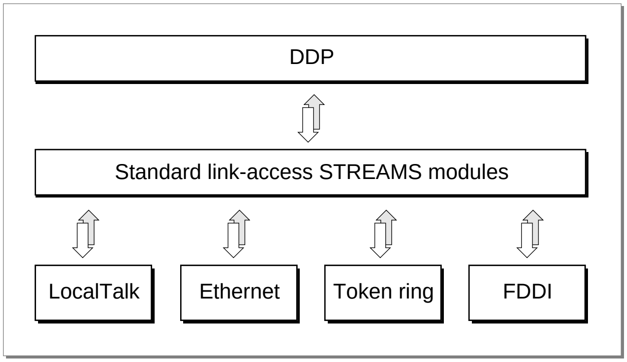
About DDP

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, *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.

Figure 15-1 shows how the DDP endpoint provider encompasses its underlying link-access STREAMS modules and its physical ports. For packets obtained from the data-link layer, DDP provides a *best-effort delivery service*.

Figure 15-1 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 receiving endpoint.
- The length of the packet matches the length specified in the packet header and does not exceed the maximum for a DDP packet.
- The checksums match (if checksumming is enabled).

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"(page 279) 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. 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 by specifying a DDP type of 0 to a specific socket, 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. If you bind another DDP endpoint to that socket, you must give it a different type.
- 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 15-1.

Table 15-1 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, the packet will also be passed to the node itself for processing. To turn this on, you set this option with a value of 1. By default this option is turned on.

You can use the `OPT_CHECKSUM` option when sending packets to enable the calculation of checksums. A value is calculated when the packet is sent. 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" (page 165).

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 `OTGetTimeStamp` function. This function gives much better resolution than the `TickCount` function.

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` functionwill return 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 defines the 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.

This option allows a multinode endpoint to tell Open Transport which of its several sockets actually sent the packet.

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. DDP uses the following Open Transport functions:

- `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.

- `OTSndUData`

  The `OTSndUData` function sends data through connectionless transactionless protocols.

  When you use this function with DDP, you can enable raw mode packet processing both on send and receive by sending a `TNetBuf` structure with the `unitdata.addr.len` field set to `0xffffffffUL`. With raw mode enabled the contents of the `unitdata.udata.buf` field is the complete DDP packet that will be sent out by the `OTSndUData` function.

  To disable raw mode packet processing, send a standard DDP packet with the `unitdata.addr` structure fields filled in normally.

- `OTRcvUData`

  The `OTRcvUData` function receives data through connectionless transactionless protocols.

- `OTRcvUDErr`

  Clears an error condition arising in the course of data transmission and returns the reason for the error.

AppleTalk Data Stream Protocol (ADSP)

Contents

This chapter describes the programming interface to Open Transport's implementation of 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 General Open Transport Functions With ADSP" (page 319) 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"(page 83).

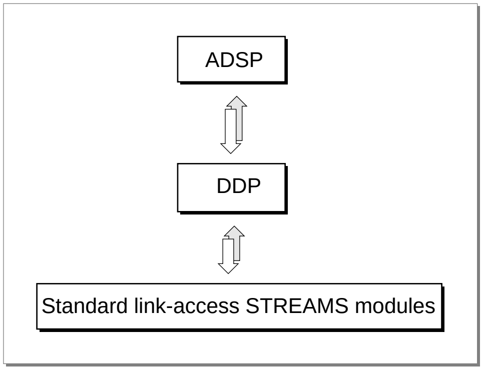
For reference information about ADSP options, see "ADSP Reference" (page 757). For an overview of ADSP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" (page 263), which also introduces and defines some of the terminology used in this chapter. ADSP under 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. Figure 16-1 shows how the ADSP endpoint provider encompasses its underlying delivery protocol and link-access Streams modules.

Figure 16-1 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. An ADSP address is a DDP address (network number and DDP socket) plus an ADSP session ID. ADSP can use the same DDP address for multiple ADSP sessions. The session ID is used to direct data to the proper ADSP session.

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. This 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 ADSP-specific features that are specific to the two AppleTalk connection-oriented protocols:

- 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

ADSP also implements a separate data channel for expedited data. This 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 and accept connections, 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 NBP 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 `qlen` field value greater than 0 during the binding process. The number you use determines how many outstanding 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 DDP 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 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. If you do not change this through the use of options, ADSP endpoints act much like other connection-oriented transactionless endpoints, and the bulk of your code would be reusable for other types of protocols (such as TCP).

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.

The 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.

If you enable the EOM option, you can send any length TSDU by setting the T_MORE flag on each send to indicate to the provider that another packet is coming that is part of this same message. When you send data without the T_MORE flag set, the provider knows this is the end of the message, and it sends an EOM packet to the remote peer. It is possible for the EOM packet to contain no data because ADSP supports the sending of zero-length packets. This is useful 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.

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);
```

Or you could call a function like that shown in Listing 7-5 (page 181) as follows:

```
err=SetFourByteOption(ep, ATK_ADSP, OPT_ENABLEEOM, 1);
```

to enable the EOM option for an ADSP endpoint.

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, since 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.

Expedited transport service data units, 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 use ETSDUs, 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 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 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.

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 a lower final value of `qlen` if it cannot handle the requested number of outstanding connection requests.

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.maxlen` 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.maxlen` 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. 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.

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. The discon->reason field contains a positive error code indicating why the connection was rejected.

AppleTalk Transaction Protocol (ATP)

Contents

This chapter describes the programming interface to Open Transport's implementation of 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 and 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 large amounts 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 General Open Transport Functions With ATP" (page 331) 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" (page 83).

For reference information about ATP options, see "ATP Reference" (page 757). For an overview of ATP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk"(page 263), 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 means of reliably 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.

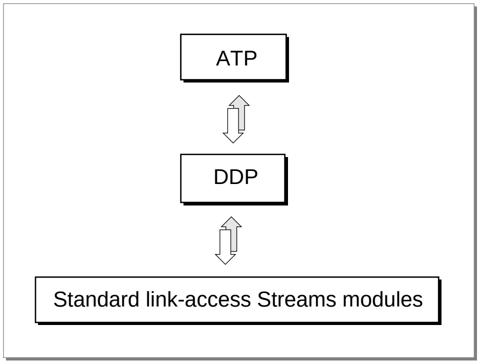
ATP is based on the concept of a transaction. In a transaction, one endpoint, called the requester, makes a request of another endpoint, called the responder, to perform a service and return a response.

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 17-1 shows how the ATP endpoint provider encompasses its underlying delivery protocol and link-access Streams modules.

Figure 17-1 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 "Specifying ATP Options" (page 329).

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.

Open Transport ATP provides XO transaction support for a request transaction when you set 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 appending data to the end of a file.

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

There are several ATP-specific options and you can use the generic Open Transport options, OPT_INTERVAL and OPT_RETRYCNT. Table 17-1 summarizes their definitions and default values. All of these options, except ATP_OPT_TRANID, can be set both globally (for the endpoint setting the option) with the OptionManagement function and locally by setting option flags for an individual transaction. The ATP_OPT_TRANID option can only be set globally.

Table 17-1 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 transaction request, 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 reply 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 for the endpoint, 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 for the endpoint, 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. You must be familiar with the descriptions of these functions in the chapter "Endpoints"(page 83) 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.

CHAPTER 18

Printer Access Protocol (PAP)

Contents

This chapter describes the programming interface to Open Transport's implementation of the Printer Access Protocol (PAP) PAP offers a connection-oriented transactionless service that has been traditionally restricted to AppleTalk printing. This chapter explains how you can use PAP to set up a printer server endpoint that awaits connection requests from active PAP endpoints. It also 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.

You should read this chapter if you want to write an application that uses PAP to print directly to AppleTalk printers or that implements a PAP server, such as a print spooler. 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 General Open Transport Functions With PAP" (page 343) 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" (page 83).

For reference information about PAP options, see "PAP Reference" (page 758). For an overview of PAP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk"(page 263), 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.

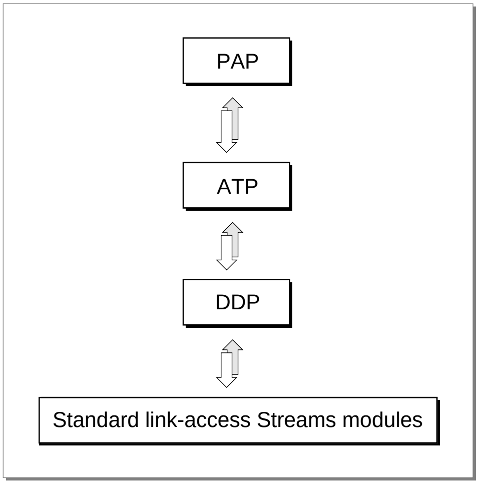
About PAP

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 printing. You use PAP when the workstation sends a print job directly to a printer connected to the network or when you send it to a print spooler, which in turn uses PAP to send it to the printer. Open Transport PAP provides a single protocol implementation that is integrated into the AppleTalk protocol stack.

Figure 18-1 shows how a PAP endpoint provider encompasses its underlying delivery protocol and link-access STREAMS modules.

Figure 18-1 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 wait time from each workstation endpoint requesting a connection. The workstations 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.

For additional information, see "Printer Access Protocol" in *Inside AppleTalk*.

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.

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 only one PAP endpoint can bind with a `qlen` greater than 0.

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 PAP-specific features:

- 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

PAP is also able to arbitrate connections requests and to accept requests from the workstations that have been waiting the longest to print.

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. 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, based on criteria that you define.

To create a passive peer that listens, you specify a `qlen` field value 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 concurrent connection between the same pair of endpoints.

Specifying PAP Options

You can use one of two options with PAP endpoints:

- The enable end-of-message options allows you to break up a data stream into discrete logical units.

  By default, PAP soes not support TSDUs.
- The open retry option allows you to retry making connection requests.

The following two sections explain the use of these options.

The End-of-Message Option

You can send a PAP data stream that contains no logical boundaries which 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.

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, set the `T_MORE` flag on each send after setting the `OPT_ENABLEEOM` option. This indicates to the provider that there are more packets coming that are part of this same message. When you send data without the `T_MORE` flag set, the provider knows this is the last packet for this message and sends an EOM packet to the remote peer. It is possible for this last (EOM) packet to contain no data because PAP supports the sending of zero-length packets. This is useful 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 of a connection, 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.

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 a PAP endpoint:

```
OTOpenEndpoint(OTCreateConfiguration("pap(EnableEOM=1)"),0, NULL, &err);
```

Or you could call a function like that shown in Listing 7-5 (page 181) as follows:

```
err=SetFourByteOption(ep, ATK_PAP, OPT_ENABLEEOM, 1);
```

to enable the EOM option for a PAP endpoint.

The Open Retry Option

By default, when a PAP endpoint provider calls the OTConnect function, which creates a connection request, it only tries to establish the connection once. This behavior is controlled by an option, PAP_OPT_OPENRETRY, whose default value is 1. (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 set a value greater than 1 for the PAP_OPT_OPENRETRY option. 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 it succeeds or until the user presses the cancel button.

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 up to 255 bytes long, the server can return that string as the server status.

The Reply Count Option

One-call specific option which you might find useful when trying to connect to some printers is the ATP_OPTREPLYCNT option.

A PAP-compliant printer will respond to a PAP connection request with a single connection reply packet in which the EOM flag is set. Some printers do not set this flag in the response packet, so ATP (upon which PAP is layered) expects additional packets to be sent. After the timeout, the endpoint resends the connection request, and again the printer responds without setting the EOM bit. The ATP_OPTREPLYCNT option tells PAP to expect only one reply packet.

Listing 18-1 demonstrates how to implement the ATP_OPTREPLYCNT option. By default when you make a connection request, the response bitmap is set to request 8 packets. By using the ATP_OPTREPLYCNT option set to 1 (with the OTConnect call), the connection request can be satisfied when the response packet is received even when the EOM bit is not set. In the sample, the endpoint is assumed to be in synchronous mode.

Listing 18-1 Using the ATP_OPTREPLYCNT option

```
OSStatus DoPAPSpecialConnect(TEndpoint* ep, UInt8 *addr, UInt32 addrLen,
                    UInt32 openRetryVal,
                    UInt32 retryIntervalVal,
                    UInt32 replyCntVal)
{
    TCall                   theCall;
    static unsigned char    optbuf[3 * kOTFourByteOptionSize];
    OSStatus                err;
    short                   i = 0;

    if (!OTIsSynchronous(ep))
    {
            /* this routines assumes that the endpoint is synchronous. */
        return (-1);
    }
```

```
        /* set the address field */
    theCall.addr.buf    = addr;
    theCall.addr.len    = addrLen;

    if (openRetryVal != 0)
    {
        ((TOption*)optbuf)->len= kOTFourByteOptionSize;
        ((TOption*)optbuf)->level= ATK_PAP;
        ((TOption*)optbuf)->name= PAP_OPT_OPENRETRY;
        ((TOption*)optbuf)->value[0]= openRetryVal;
        (((TOption*)optbuf) + 1)->len= kOTFourByteOptionSize;
        (((TOption*)optbuf) + 1)->level= ATK_PAP;
        (((TOption*)optbuf) + 1)->name= OPT_INTERVAL;
        (((TOption*)optbuf) + 1)->value[0]= retryIntervalVal;

        i += 2; // increment the options counter index
    }

    if (replyCntVal != 0)
    {
            /* If the replyCntVal is non zero then make sure that it */
            /* falls between 1 and 8 */
        if (replyCntVal < 1)
            replyCntVal = 1;
        if (replyCntVal > 8)
            replyCntVal = 8;

        (((TOption*)optbuf) + i)->len= kOTFourByteOptionSize;
        (((TOption*)optbuf) + i)->level= ATK_ATP;
        (((TOption*)optbuf) + i)->name= ATP_OPT_REPLYCNT;
        (((TOption*)optbuf) + i)->value[0]= replyCntVal;

        i++;/* increment the options counter index */
    }

        /* go ahead and set the option buffer field */
    theCall.opt.buf= (UInt8*)optbuf;
        /*  set the length field depending on number of options set. */
    theCall.opt.len= i * kOTFourByteOptionSize;

    theCall.udata.len= 0;
    err = ep->Connect(&theCall, NULL);

    return err;
}
```

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. You must be familiar with the function descriptions in the chapter "Endpoints"(page 83) 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 a lower final value of `qlen` if it cannot handle the requested number of outstanding connection requests.

OTConnect

The `OTConnect` function requests 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.maxlen` 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.maxlen` 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. 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.

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.maxlen field to 0. PAP ignores the discon->udata.buf field.

CHAPTER 19

Serial Endpoint Providers

Contents

This chapter describes how you can use serial endpoint providers to transfer data between devices connected to a serial 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.

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. For information about locating all the serial ports available through Open Transport, see "Ports" (page 191).

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" (page 360) 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" (page 357) describes the options you can specify when you configure a serial endpoint provider. The Serial Reference Chapter (page 763)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 serial data transfers through the built-in serial port and any serial choice registered with the Communications Resource Map. 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.

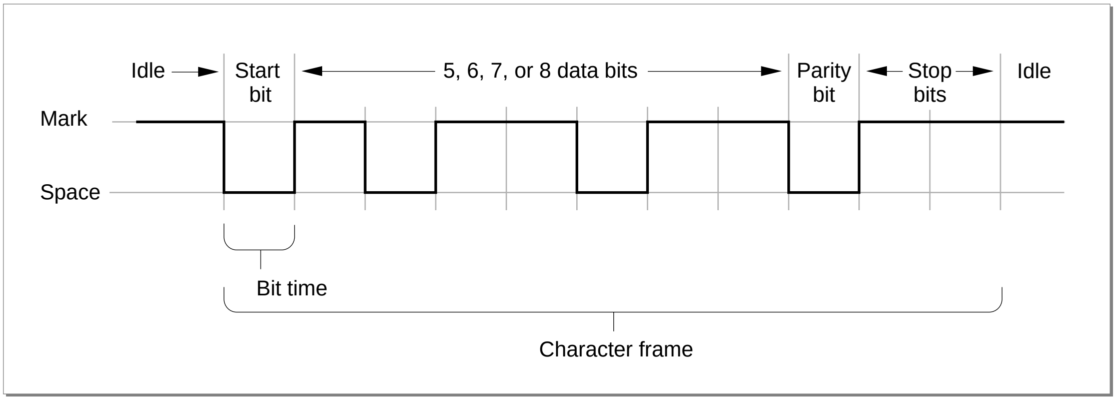
About Serial Communication

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**. Changing the state of the line by shifting the voltage is called a **space.** The receiver interprets a space as a 0 bit, and a mark as a 1 bit. These transitions are shown in Figure 19-1.

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 state at periodic intervals, known as the **bit time,** to determine whether a 0 bit or a 1 bit is present on the line.

Figure 19-1 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 commonly 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 Association (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.** These two signals are connected to each other. Note that in the definitions of these signals which follow, 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 Clear To Send (CTS) signal indicates that the DCE (your modem or printer) is ready to send data.

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 peers agree on a clocking mechanism before data is transferred; in **synchronous communication**, the signal carries the clocking information. 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.

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 deasserts 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" (page 83).

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. The configuration string you supply depends on which serial port you want to open. The following constants are defined for the built-in ports; you can find other serial ports using the techniques descripted in "Ports" (page 191).

Constant name	String value	Description
kSerialName	"serial"	Default serial port
kSerialPortAName	"serialA"	Serial port A (printer port)
kSerialPortBName	"serialB"	Serial port B (modem port)
kSerialPortABName	"serialAB"	Serial port AB (combined printer/modem port)

For example, the following line of code opens a serial endpoint on serial port A:

```
OTOpenEndpoint(OTCreateConfiguration(kSerialPortAName));
```

There may be other serial ports available, such as those registered by the Communications Toolbox.

To close a serial endpoint provider, you use the standard Open Transport function OTCloseProvider, described in the chapter "Providers" (page 61).

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 the OTSnd and OTRcv functions to be called. 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, that has a queue length of 0 or 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 on the accepting endpoint 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 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 Reference" (page 383), accesses the low-level serial driver control and status functions.

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 eight 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 options is more complicated, and their use is described in the two subsequent sections.

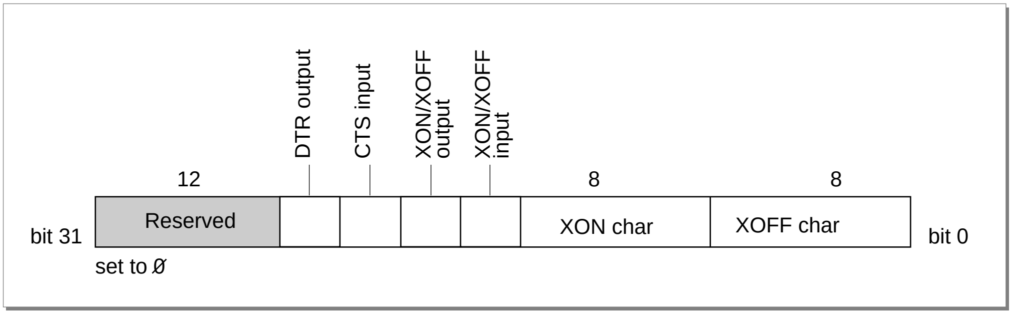
- The baud rate option sets the serial baud rate.The serial module chooses the closest baud rate supported that matches the requested rate. Possible values range from 300 to 230.4K baud transmission rates (depending on the hardware capability). The default value is 19200 baud.
- The data bits option 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 and delivering the data is already has . The default value is 10.
- 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 macros (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.

Controlling Serial Port I/O Handshaking

You can use the SRL_OPT_HANDSHAKE option to customize serial port handshaking in a variety of ways. For instance, you can request that an input handshake be controlled by the CTS line, or by the XON/XOFF sequence. To control the handshaking behavior, you pass in a 4-byte usigned integer value with the SRL_OPT_HANDSHAKE option.

A schematic diagram of this 4-byte option value is shown in Figure 19-2

Figure 19-2 Serial port I/O handshaking



The high word (16 bits) of the integer is a bitmap with one 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 is the XON character value; the lowest byte is the XOFF character. If these values are 0 and XON/OFF handshaking is requested, the default values of control-S for XOFF and control-Q for XON are used. The default value of this option is no handshaking.

Open Transport provides a macro and a C++ inline function (OTSerialHandshakeData) that you can use to create the 4-byte option value.

For example, to enable XON/XOFF input handshaking, but to specify that the XON character be control-T rather than control-Q, you can create an option structure as follows:

```
opt.value = OTSerialHandshakeData( kOTSerialXOnOffInputHandshake |
                                   kOTSerialXOnOffOutputHandshake,
                                           'T' - 64, 'S' - 64);
```

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 occurred:

- 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.
- 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 1024 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.

Open Transport serial services are layered on top of the serial hardware driver. The capabilities of Open Transport endpoints depend on the driver. Consult the hardware documentation to determine the limitations of Open Transport for serial endpoints.

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 on 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"(page 83) 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`	Connection oriented transactionless service. 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 use the `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, and `OTGetEndpointInfo` functions to obtain the current values for these parameters. ▲

These fields and the significance of their values are described in more detail in the chapter "Endpoints Reference"(page 421).

Using Endpoint Functions With Serial Endpoints

This section describes serial-specific information about functions described in the chapter "Endpoints" (page 83).

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 make connections, the endpoint can have a value of 0 or 1. A value greater than 1 results in an 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 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.maxlen 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.maxlen` field to 0. Serial endpoints ignore the `discon->udata.buf` field.

PART TWO

Open Transport Reference

PART ONE

Initializing and Closing Open Transport Reference

Contents

This chapter describes the contents, data types, and functions you use to initialize and close Open Transport, to configure a provider, and to specify an address. The chapter "Open Transport"(page 5) explains their use.

This section describes the basic configuration management constants, the Gestalt function selector and response bits, and the configuration structure for Open Transport.

Error Constants

If Open Transport is unable to create an OTConfiguration structure (page 398), it returns one of the following values, depending on whether the specified path was invalid or whether there was insufficient memory to create the structure:

```
#define kOTInvalidConfigurationPtr ((OTConfiguration*)-1L)

#define kOTNoMemoryConfigurationPtr ((OTConfiguration*)0)
```

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 'ota kn' and 'otvr' selectors.

The 'otvr' selector determines the Open Transport version in NumVersion format. For more information on Apple's version numbering scheme and the NumVersion format, see *Technote OV12:Version Territory.* The 'otvr' selector was not implemented until version 1.1 of Open Transport. The absence of this selector does not mean that Open Transport is not present. You can also check for the availability of Open Transport by calling the function InitOpenTransport (page 373).

The 'otan' selector 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 {
    gestaltOpenTptVersions              = 'otvr',
    gestaltOpenTpt                      = 'otan',
    gestaltOpenTptPresentMask           = 0x00000001,
    gestaltOpenTptLoadedMask            = 0x00000002,
    gestaltOpenTptAppleTalkPresentMask  = 0x00000004,
    gestaltOpenTptAppleTalkLoadedMask   = 0x00000008,
    gestaltOpenTptTCPPresentMask        = 0x00000010,
    gestaltOpenTptTCPLoadedMask         = 0x00000020,
    gestaltOpenTptIPXSPXPresentMask     = 0x00000040,
    gestaltOpenTptIPXSPXLoadedMask      = 0x00000080,
    gestaltOpenTptPresentBit            = 0,
    gestaltOpenTptLoadedBit             = 1,
    gestaltOpenTptAppleTalkPresentBit   = 2,
    gestaltOpenTptAppleTalkLoadedBit    = 3,
    gestaltOpenTptTCPPresentBit         = 4,
    gestaltOpenTptTCPLoadedBit          = 5,
    gestaltOpenTptIPXSPXPresentBit      = 6,
    gestaltOpenTptIPXSPXLoadedBit       = 7
};
```

For more information about the `Gestalt` function, see *Inside Macintosh: Operating System Utilities.*

The OTConfiguration Structure

Open Transport functions that open a provider take as a parameter a pointer to a configuration structure that specifies the configuration of that provider. For example, the configuration structure of an endpoint specifies which protocol modules the endpoint uses. To create a configuration structure and obtain a pointer to it, you call the `OTCreateConfiguration` function (page 376). To make a copy of a configuration structure, you call the `OTCloneConfiguration` function (page 378).

The contents of the `OTConfiguration` structure are private and so is the `OTConfiguration` data type that defines it.

```
struct OTConfiguration;
typedef struct OTConfiguration OTConfiguration;
```

See “Error Constants” (page 397) for values that can be returned if the configuration was not successful.

The OTAddress Structure

Addresses in Open Transport all begin with a common structure, which is followed by fields that are protocol-specific. The common structure is defined by the `OTAddress` type:

```
struct OTAddress
    {
        OTAddressType   fAddressType;
        UInt8           fAddress[1];
    };
typedef struct OTAddress OTAddress;
```

The `OTAddress` type itself is abstract. You would not declare a structure of this type because it does not contain any address information. However address formats defined by Open Transport protocols all use the `fAddressType` field to describe the format of the fields to follow, which do contain address information. For an example of how this data type is used in creating an address, see the section “Addressing in Open Transport” (page 37).

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 use a `TNetbuf` structure as an input parameter, you specify the location and size of a buffer containing information you want to send. If you use a `TNetbuf` structure as an output parameter, you specify the location and the maximum size of the buffer used to hold information when the function returns.

You use a `TNetbuf` structure to describe the location and size of contiguous data. Open Transport allows you to describe noncontiguous data with the `OTData` structure. For more information, see “Advanced Topics Reference”(page 673)

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

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

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 the functions you use to initialize and close Open Transport and to create, clone, and delete a configuration structure.

Initializing and Closing Open Transport

Open Transport provides three functions that you can use to initialize and close Open Transport.

InitOpenTransport

Initializes the parts of Open Transport for use by the application or code resource.

C INTERFACE

```
OSStatus InitOpenTransport(void);
```

C++ INTERFACE

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

PARAMETERS

function result An error code. See Appendix B.

DISCUSSION

Call this function before using other Open Transport functions.

If you need to know whether Open Transport is present at start-up time, but you don’t need to use it (e.g. an installer) ,you can use the `Gestalt` function and its Open Transport selectors as described on (page 369). However, for normal applications, your calling the `InitOpenTransport` function is sufficient to test for the presence of Open Transport.

To initialize only the parts of Open Transport that handle ports and implement the Open Transport utility functions call the `InitOpenTransportUtilities` function (page 374).

SPECIAL CONSIDERATIONS

You must make sure that your A5 world is correctly initialized for 68000 code resources.

If your program uses the Apple Shared Library Manager (ASLM), you must call the `InitLibraryManager` function to initialize ASLM before calling the `InitOpenTransport` function. To initialize ASLM, use the `InitLibraryManager` function, described in the *Apple Shared Library Manager Developer’s Guide*.

For applications, Open Transport patches the `ExitToShell` trap when you call the function `InitOpenTransport`. The patch calls the function `CloseOpenTransport` if Open Transport is still active when your application quits.

SEE ALSO

"The Gestalt Selector and Response Bits" (page 369).

"Initializing and Closing Open Transport"(page 31).

The `CloseOpenTransport` function (page 375).

InitOpenTransportUtilities

Initializes only that part of Open Transport that handles ports and implements Open Transport utility functions.

C INTERFACE

```
OSStatus InitOpenTransportUtilities(void);
```

C++ INTERFACE

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

PARAMETERS

function result A return value other than `kOTNoError` indicates that the Open Transport is not installed.

DISCUSSION

If you have called the function `InitOpenTransport`, you do not need to call the `InitOpenTransportUtilities` function.

Call the `InitOpenTransportUtilities` function before calling Open Transport functions that manipulate ports or before calling Open Transport utility functions.

SPECIAL CONSIDERATIONS

If your program uses the Apple Shared Library Manager (ASLM), you must call the `InitLibraryManager` function to initialize ASLM before calling the `InitOpenTransportUtilities` function.

SEE ALSO

The `InitOpenTransport` function (page 373).

The `Gestalt` function, described in *Inside Macintosh: Operating System Utilities.* "The Gestalt Selector and Response Bits" (page 369).

"Initializing and Closing Open Transport (page 31).

The `CloseOpenTransport` function (page 375).

CloseOpenTransport

Unregisters your application or code resource connection to Open Transport.

C INTERFACE

```
void CloseOpenTransport(void);
```

C++ INTERFACE

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

DISCUSSION

The `CloseOpenTransport` function tells Open Transport that your application or code resource has finished using it. You can call this function only at system task time. You must not call this function if you have any outstanding network I/O in progress, such as an outstanding asynchronous operation.

When your application finishes using Open Transport, you have the option of using this function to unload Open Transport without stopping execution if your application has other tasks to perform that do not require Open Transport. If you are writing an application, you are not required to use this function, but it is *strongly* recommended that you do so.

If you are writing a code resource, a CFM code fragment, or a shared library, you must call the `CloseOpenTransport` function before unloading from memory.

System software cannot unload the Open Transport kernel until the last software module on your computer that called the `InitOpenTransport` function has also called the `CloseOpenTransport` function.

SPECIAL CONSIDERATIONS

If your client uses the Apple Shared Library Manager, be sure you call the `CleanupLibraryManager` function *after* calling the `CloseOpenTransport` function.

SEE ALSO

The `InitOpenTransport` function (page 373).

The `InitOpenTransportUtilities` function (page 374).

"Getting Started With Open Transport" in *Networking With Open Transport*.

Creating, Cloning, and Disposing of a Configuration Structure

This section describes the Open Transport functions you can use to create, clone, and destroy a provider configuration structure.

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.

function result A pointer to a private `OTConfiguration` structure. If the value specified with the path parameter was invalid, the function returns the result `kOTInvalidConfigurationPtr`. If there was not enough memory to create the structure, the function returns the result `kOTNoMemoryConfigurationPtr`.

DISCUSSION

The `OTCreateConfiguration` function creates a configuration structure that defines the software modules, hardware ports, and options that Open Transport uses when you open a provider. This is a private structure, defined by the `OTConfiguration` data type (page 370). To create a configuration structure, use the `path` parameter to pass a string describing the provider service desired to the `OTCreateConfiguration` function.

The simplest possible value of the `path` parameter is the name of the highest-level protocol you want to use; for example, "tcp" . If you do not specify a complete communications path, Open Transport 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 Data Stream Protocol (ADSP) protocol module layered above the Datagram Delivery Protocol (DDP) protocol module, which is, in turn, layered on the default port configured in the AppleTalk control panel.

If you want to identify a particular port in the configuration string, you use the port name to do so, described in the chapter "Ports"(page 191). More typically, however, you leave this value blank—for example, using a string with only "adsp" , which configures the provider with whatever port is specified in the AppleTalk 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 DDP checksum option is enabled. For a list of valid options for each protocol, see the corresponding protocol reference chapter.

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. If the function returns an error code, it is safe to pass the error code as the `OTOpenEndpoint` `OTConfiguration`* parameter. `OTOpenEndpoint` will return the appropriate error.

SPECIAL CONSIDERATIONS

Functions that open providers dispose of 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

"Reusing Provider Configurations" (page 37) .

The `OTCloneConfiguration` function(page 378).

The `OTDestroyConfiguration` function (page 379).

The `OTOpenEndpoint` function (page 437) . The `OTAsyncOpenEndpoint` function(page 438).

OTCloneConfiguration

Copies an `OTConfiguration` structure.

C INTERFACE

```
OTConfiguration* OTCloneConfiguration(OTConfiguration* cfig);
```

C++ INTERFACE

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

PARAMETERS

`cfig`	A pointer to the `OTConfiguration` structure that you want to copy.
function result	A pointer to a private `OTConfiguration` structure.

DISCUSSION

The `OTCloneConfiguration` function copies the `OTConfiguration` structure that you specify in the `cfig` parameter and returns a pointer to the copy. Because the format of an `OTConfiguration` structure is private, you must use the `OTCloneConfiguration` function to obtain two identical structures. Creating configurations through cloning is much faster than recreating them using the `OTCreateConfiguration` function.

SEE ALSO

"Initiating and Closing Open Transport"(page 31).

The `OTCreateConfiguration` function (page 376).

The `OTDestroyConfiguration` function (page 379).

OTDestroyConfiguration

Disposes of an `OTConfiguration` structure.

C INTERFACE

```
void OTDestroyConfiguration(OTConfiguration* cfig);
```

C++ INTERFACE

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

PARAMETERS

`cfig` A pointer to the `OTConfiguration` structure that you want to dispose of.

DISCUSSION

The `OTDestroyConfiguration` function disposes of the `OTConfiguration` structure that you specify in the `cfig` parameter and releases all associated memory.

Functions that open providers automatically disposes of 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 dispose of an `OTConfiguration` structure that you have not actually used to open a provider.

SEE ALSO

"Initiating and Closing Open Transport"(page 31).

The `OTCreateConfiguration` function (page 376).

The `OTCloneConfiguration` function (page 378).

CHAPTER 21

Providers Reference

Contents

This chapter describes general provider constants, data types, and functions that you can use with providers of any type. For conceptual information about providers, see "Providers" (page 61).

Constants and Data Types

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

Error-Checking Constant

Open Transport provides a constant that you can use to initialize a pointer to the opaque provider reference structure. You can then check that the current provider reference is valid before passing it to provider functions. The constant is defined as follows:

```
#define kOTInvalidProviderRef((ProviderRef)(0)
```

Event Codes

Your application can include a notifier function that the provider calls to inform you that some provider event has occurred. The provider passes an event code for the function's `code` parameter. The event code specifies which provider event 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. See "Handling Events for Endpoints" (page 102) for more information.

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,
    kOTSyncIdleEvent
    kOTProviderIsReconnected        = (OTEventCode)0x23000002.
    kOTProviderWillClose            = (OTEventCode)0x24000001,
    kOTProviderIsClosed             = (OTEventCode)0x24000002,
};
```

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 connection 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 type of service of the endpoint 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 will not get another indication that data has arrived until you have read all the data and got this error.

T_ERROR
Obsolete.

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 will not get another indication that data has arrived until you have read all the data and got this error.

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.

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 type of service for the endpoint 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 type of service of the endpoint 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 resRef parameter to 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 it 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 resRef parameter to the OTAccept function.

T_REPLYCOMPLETE
The OTSndUReply or OTSndReply functions have completed. The cookie parameter contains the sequence number used in the OTSndUReply or OTSndReply call.

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 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 Open Transport is done using the buffers containing the data you are sending. If you called the `OTSnd` function, the `cookie` parameter contains the `buf` parameter. If you 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.

You should not wait for the `T_MEMORYRELEASED` event from a previous send operation to trigger more sends. The exact time this event occurs depends on how the underlying provider is implemented. It might hold on to memory until the next send occurs, or behave in some other way which causes it to delay releasing memory.

Note that `T_MEMORYRELEASED` events can reenter your notifier. See “OTAckSends” (page 401) for more information.

`T_REGNAMECOMPLETE` The `OTRegisterName` function has completed. The `cookie` parameter is the `reply` parameter, unless it was `NULL`. In this case, it is the `req` parameter.

`T_DELNAMECOMPLETE` The `OTDeleteName` function or the `OTDeleteNameByID` function has completed. The `cookie` parameter contains the `name` parameter or the `nameId` 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.

`kOTSyncIdleEvent` A synchronous call is waiting to complete. Call the function `YieldToAnyThread` from your notifier to allow other concurrent processes to obtain processing time. See “Using Synchronous Processing With Threads” (page 130) for more information.

`kOTProviderIsDisconnected`

Your provider was bound with a `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 result code specifying the reason why the provider had to close. For example, the user decided to switch links using the TCP/IP or AppleTalk control panel. The result codes that can be returned are in the range –3280 through –3285; these are documented in "Result Codes" (page 785).

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. After this, 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.

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 "Initializing Open Transport" (page 31).

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 the various type of providers. When you finish using a provider of any type, always call the OTCloseProvider function to close and delete the provider.

You can also transfer ownership of a provider using the OTTransferProviderOwnership function, and the OTWhoAmI function, to obtain the ID of the current client.

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(
                     OSStatus* errPtr);
```

PARAMETERS

ref	The provider reference for the provider to be transferred.
prevOwner	The previous owner of the provider.
errPtr	A pointer to a result code.
function result	A new provider reference that replaces the original.

DISCUSSION

The OTTransferProviderOwnership function transfers the ownership of the provider indicated by the ref parameter from whoever created the provider, specified by the prevOwner parameter, to the current Open Transport client. The new owner must obtain the client ID and the provider `ref` from the current owner before calling this function. You must use this function whenever some other entity (typically a shared library) opens an endpoint on behalf of the client.

Under Open Transport, a provider allocates a small amount of memory from the client's heap. In addition, Open Transport automatically cleans up behind clients that call the `CloseOpenTransport` function. If a shared library creates a provider on your behalf and that shared library subsequently unloads while you are still using the provider, two things will happen: The memory for the provider is no longer valid, and the provider is closed. By using the `OTTransferProviderOwnership` function, you can gain ownership of the provider. Once you do, Open Transport allocates a new provider reference in your memory pool and the provider is marked as belonging to the calling client. The old provider reference is then invalid and should not be used.

SPECIAL CONSIDERATIONS

When installing a notifier for a provider, Open Transport assumes that the `OTNotifyProcPtr` pointer is in the same architecture as the call being made. After transferring ownership, remove any already installed notifiers and install your own, unless you are sure that the provider's existing notifier is of the correct architecture and makes sense for you

▲ **WARNING**
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

The `OTWhoAmI` function (page 391).

OTWhoAmI

Returns the current client's ID.

C INTERFACE

```
OTClient OTWhoAmI(void);
```

C++ INTERFACE

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

PARAMETERS

function result A 32-bit value specifying the client ID.

DISCUSSION

In transferring provider ownership from an old client to a new client, the old client must obtain its own current client ID (by calling the `OTWhoAmI` function) and must use some means to communicate this value to the new client. The new client then uses this value as the `prevOwner` parameter to the `OTTransferProviderOwnership` function that it calls to gain ownership of the provider.

SEE ALSO

The `OTTransferProviderOwnership` function (page 390).

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.
function result	An error code. See Appendix B for more information.

DISCUSSION

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

If there are outstanding asynchronous calls when you close the provider, these calls will never complete. If there are outstanding synchronous calls, the calls will complete with the result of `kOTCanceledErr`.

The blocking/nonblocking status of the provider governs what actually happens when a provider is closed. In nonblocking mode, closing a provider flushes all outgoing commands in the stream before immediately closing the provider. In blocking mode, the stream is given up to 15 seconds per module to allow outgoing commands and data to be processed before closing the stream. Insofar as the caller is concerned, the provider is closed immediately.

▲ **WARNING**

You need to be sure that there are no outstanding `T_MEMORYRELEASED` 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

For information about opening different types of providers, please see the appropriate reference chapter in this book.

Controlling a Provider’s Modes 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 find out the current setting 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 affects how it interacts with a client. In servicing a client's request, a provider might be able to deal with the request immediately, or it might have to queue the request and deal with it later. 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" (page 72).

You can use the `OTAckSends` and `OTDontAckSends` functions to specify whether a provider acknowledges sends. This determines whether a provider copies data that you send . 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 to synchronous mode.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider which you want to set to synchronous mode.
function result	An error code. See Appendix B for more information.

DISCUSSION

The `OTSetSynchronous` function causes all provider functions to run synchronously when using the provider that you specify. Changing this does not affect its notifier function, if any is installed for this provider; the notifier function remains installed.

You can call this function at any time and it will always succeed; however, you should take care changing this while outstanding asynchronous calls are in progress. See "Specifying How Provider Functions Execute" (page 71) for more information.

To place a provider in asynchronous mode, call the `OTSetAsynchronous` function, (page 394). To find out a provider's mode of execution, call the `OTIsSynchronous` function (page 396).

SEE ALSO

"Specifying How Provider Functions Execute" (page 71).

OTSetAsynchronous

Sets a provider to asynchronous mode.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref` The provider reference of the provider which you want to set to asynchronous mode..

function result An error code. See Appendix B for more information.

DISCUSSION

The `OTSetAsynchronous` function causes all functions for the provider specified in the `ref` parameter to run asynchronously. You should install a notifier function for the provider to receive completion and other events. You can install a notifier function either before or after calling the `OTSetAsynchronous` function.

You can call this function at any time and it will always succeed.

To set a provider to synchronous mode, call the `OTSetSynchronous` function (page 394). To find out a provider's current setting, call the `OTIsSynchronous` function (page 396).

SEE ALSO

"Specifying How Provider Functions Execute" (page 71).

"Using Notifier Functions" (page 405).

OTIsSynchronous

Determines whether a provider is asynchronous.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

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

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

DISCUSSION

To set a provider to synchronous mode, call the `OTSetSynchronous` function (page 394). To set a provider to asynchronous mode, call the `OTSetAsynchronous` function (page 395).

OTCancelSynchronousCalls

Cancels any currently executing synchronous function for a specified provider.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

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

`err` The error code value to be returned to the synchronous caller, typically `kOTCanceledErr`.

function result For the C interface, `kEBADFErr` is returned if the reference is invalid. The function can also return `kOTOutStateErr` if the endpoint is closed.

DISCUSSION

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 to prevent a synchronous function from hanging the system. You can make the call at interrupt time by installing a Time Manager task that executes after a given amount of time has passed. You can also call this function from the provider's notifier after getting a `kOTSyncIdleEvent` after a timeout.

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

Timer tasks are described in the chapter "Utitlities Reference."

"Specifying How Provider Functions Execute" (page 71).

OTSetBlocking

Sets a provider to wait or block until it can complete a function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider that is to block.
function result	An error code. See Appendix B for more information.

DISCUSSION

For a full description of the effect of this call, see “Setting a Provider’s Blocking Status” (page 72).

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 data.

To set a provider’s blocking mode to nonblocking, call the `OTSetNonBlocking` function (page 399). To find out a provider’s current blocking mode, call the `OTIsNonBlocking` function (page 400).

OTSetNonBlocking

Sets a provider to return with an error if it cannot immediately complete a function.

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.
function result	An error code. See Appendix B for more information.

DISCUSSION

For a full description of the effect of this call, see "Setting a Provider's Blocking Status" (page 72).

The `OTSetNonBlocking` function causes provider functions to return a result code immediately, instead of waiting for a function 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 closes the provider. Conversely, in blocking mode, the provider would give each STREAMS module up to 15 seconds to deal with outgoing data.

To set a provider's blocking status to blocking, call the `OTSetBlocking` function, (page 398). To find out a provider's current blocking mode, call the `OTIsNonBlocking` function (page 400).

SEE ALSO

"Setting a Provider's Blocking Status" (page 72).

OTIsNonBlocking

Returns a provider's current blocking mode.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

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

function result `True` if the provider is currently in nonblocking mode or `false` if it is in blocking mode.

DISCUSSION

To set a provider's blocking status to blocking, call the `OTSetBlocking` function (page 398). To set a provider's blocking status to nonblocking, call the `OTSetNonBlocking` function (page 399).

SEE ALSO

"Setting a Provider's Blocking Status" (page 72).

OTAckSends

Specifies that a provider not make an internal copy of data being sent.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider that is sending data.
function result	An error code. See Appendix B for more information.

DISCUSSION

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 the `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. The `T_MEMORYRELEASED` event is not serialized with other notification events and will re-enter your notifier function.

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` functions, the `OTAckSends` function returns a `kOTChangeStateErr` message.

Although you must wait for a `T_MEMORYRELEASED` event before you touch the buffer containing the data being sent, you do not have to wait for a `T_MEMORYRELEASED` event to send more data. You can continue to process data normally, sending data until you get a flow error and then resuming sends when you receive the `T_GODATA` event.

▲ **WARNING**
You need to be sure there are no outstanding `T_MEMORYRELEASED` 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. ▲

To find out a provider's current send-acknowledgment mode, call the `OTIsAckingSends` function (page 404).

The send-acknowledgment mode of a provider is ignored by mapper providers, AppleTalk service providers, and TCP/IP service providers.

COMPLETION EVENT CODES

`T_MEMORYRELEASED`

SPECIAL CONSIDERATIONS

Because of the complexity of handling flow control, Open Transport performance suffers when the acknowledge sends option is used with noncontiguous data, such as when you pass an `OTData` structure to the `OTSnd` function. It is best to use this option with contiguous data or with `OTData` structures whose last element is large.

SEE ALSO

The `OTDontAckSends` function(page 403).

"Setting a Provider's Send-Acknowledgment Status" (page 72).

See the chapter "Advanced Topics" for more information about acknowledging sends.

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.
function result	An error code. See Appendix B for more information.

DISCUSSION

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` functions, the `OTDontAckSends` function returns a `kOTChangeStateErr` message.

To find out whether a provider is acknowledging sends, call the `OTIsAckingSends` function (page 404).

SEE ALSO

"Setting a Provider's Send-Acknowledgment Status" (page 72).

OTIsAckingSends

Determines whether a provider copies data when sending it.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The provider reference of the provider sending data.
function result	`True` if the provider does not copy sent data and `false` if it does.

DISCUSSION

To specify that a provider not copy data on sends, call the `OTAckSends` function (page 401). To specify that a provider copy data on sends, call the `OTDontAckSends` function (page 403).

SEE ALSO

"Setting a Provider's Send-Acknowledgment Status" (page 72)

Using Notifier Functions

To receive notice of provider events, you must install a notifier function. If the provider was opened synchronously, you do this by calling the `OTInstallNotifier` function. If the provider was opened asynchronously, 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.

You can use the `OTUseSyncIdleEvents` function to send idle events to your notifier.

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. (This is a function pointer not a universal proc pointer, consequently no mixed mode glue is required. You must never pass a universal procedure pointer for this parameter.)

For C++ applications, the `proc` parameter may point to either a C function or, depending on your development environment, 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. If you don't need context information, pass `NIL` for this parameter.

function result An error code. See Appendix B for more information.

DISCUSSION

The `OTInstallNotifier` function installs a notifier function for the provider that you specify. The notifier function remains installed until you remove it using the `OTRemoveNotifier` function or until you close the provider. (Setting a provider to synchronous mode, does not remove the notifier.)

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.

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.

You should not call the `OTInstallNotifier` function when 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 `OTInstallNotifier` function returns with the `kOTAccessErr` result.

IMPORTANT

For 68000-based code, 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

"Application-Defined Notifier Functions" (page 413).

OTRemoveNotifier

Removes a provider's notifier function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

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

DISCUSSION

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

SEE ALSO

"Application-Defined Notifier Functions" (page 413).

The `OTInstallNotifier` function (page 405).

OTEnterNotifier

Limits the notifications that can be sent to your notifier. (This function can only be called for Open Transport version 1.1.1 or later.)

C INTERFACE

```
Boolean OTEnterNotifier (ProviderRef ref);
```

C++ INTERFACE

```
Boolean TProvider::EnterNotifier();
```

PARAMETERS

`ref` A provider reference for the provider that must not be interrupted by asynchronous and completion events.

function result The function returns false if you have already called `OTEnterNotifier` on this provider or if you make the call from within your notification routine.

DISCUSSION

This function prevents Open Transport from sending the specified provider an asynchronous or completion event. After making this call, notification is subject to the same rules as if you were already executing within your notification routine. This allows you to determine whether or not you should call the `OTLeaveNotifier` function.

You can use calls to `OTEnterNotifier` and `OTLeaveNotifier` to bracket critical sections of your code and to minimize or eliminate synchronization problems.

To have Open Transport resume sending primary and completion events, you need to call the `OTLeaveNotifier` function (page 409).

SEE ALSO

"Using Asynchronous Processing With a Notifier" (page 134).

OTLeaveNotifier

Allows Open Transport to resume sending primary and completion events. (This function can only be called for Open Transport version 1.1.1 or later.)

C INTERFACE

```
void OTLeaveNotifier (ProviderRef ref);
```

C++ INTERFACE

```
void TProvider::LeaveNotifier();
```

PARAMETERS

`ref` A provider reference for the provider that can again be interrupted by asynchronous and completion events.

DISCUSSION

This function allows Open Transport to send the specified provider an asynchronous or completion event. You should call this function after having called the `OTEnterNotifier` function (page 408), to reenable processing of these events.

SEE ALSO

"Using Asynchronous Processing With a Notifier" (page 134).

OTUseSyncIdleEvents

Allows synchronous idle events to be sent to your notifier. (This function can only be called for Open Transport version 1.1.1 or later.)

C INTERFACE

```
OSStatus OTUseSyncIdleEvents (
                        ProviderRef ref,
                        Boolean onOff);
```

C++ INTERFACE

```
OSStatus TProvider::UseSyncIdleEvents(Boolean onOff);
```

PARAMETERS

`ref`	A provider reference for the provider whose notifier function is to receive synch idle events.
`onOff`	A Boolean value indicating whether synchronous idle events can be sent to your notifier.
function result	An error code. See Appendix B for more information.

DISCUSSION

This function facilitates the interaction between a notifier and the Thread Manager. Its primary purpose is to allow scheduling of other threads while Open Transport is waiting for a synchronous call to complete. This is the only time that scheduling of a thread from within a notification function is legal.

After you call this function with a value of `true`, Open Transport will generate `kOTSyncIdleEvent` events and send them to your notifier whenever your execution thread is blocked waiting for an operation to complete. At this point, it is safe to use the `YieldtoThread` function. If performance is not an issue, this is a handy method for handling threads. But because the `YieldtoThread` function will eventually cause the Thread Manager to switch to a thread that calls `WaitNextEvent`, this can impact your performance.

SEE ALSO

The chapter “Programming With Open Transport.”

Sending Module-Specific Commands

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

▲ **WARNING**
Sending module-specific commands is for advanced users only. It can give rise to conflicts for the provider and confuse the endpoint library. ▲

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.

function result See Discussion.

DISCUSSION

The `OTIoctl` function sends a module-specific command to an Open Transport protocol module.

If the provider is in synchronous mode, the function waits until the command completes. It then returns 0 or a positive integer. The positive integer's meaning is defined by the Open Transport module that you are using. Refer to the documentation for that module. If the `OTIoctl` function fails while executing synchronously, its return value is a negative integer corresponding to an Open Transport result code.

If the provider is in asynchronous mode, 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 installed for the provider, 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.

IMPORTANT

You can have only one oustanding call to the `OTIoctl` function at any one time. ▲

COMPLETION EVENT CODES

`kStreamIoctlEvent`

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.

Application-Defined Notifier 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.

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

MyNotifierCallbackFunction

After you install a notifier function on a provider, the provider calls the notifier function each time a provider event occurs. The declaration for the procedure is

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

C INTERFACE

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

C++ INTERFACE

None. C++ appplications use the C interface to this 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.

`code` An event code indicating the event that occurred. Possible values for event codes are given in the event code enumeration (page 383).

`result` For completion events, the result code of the completed provider function, identified by the `code` parameter. For asynchronous 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 event code.

`cookie` A pointer to data. The meaning and type of the data vary, depending on the event code returned in the `code` parameter. For additional information, see the event codes enumeration (page 383).

DISCUSSION

Using a notifier function is the recommended way for your application to handle provider events. After you install a notifier function for a provider, the function is called by the provider each time a provider 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.

To install a notifier function for an existing provider, call the `OTInstallNotifier` function (page 405). You can also install a notifier 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.

You can install the same notifier function for two or more providers and use the `contextPtr` parameter to distinguish among them. Typically the data structure referenced by the `contextPtr` parameter includes a provider reference or some other identifier that uniquely identifies the provider for which the notifier is called.

Open Transport attempts to minimize re-entrancy and stack overflow problems by queueing calls to a notification routine, but this behavior is not guaranteed. Open Transport does guarantee that the notification routine will not be reentered if you are processing a completion event (the event code ends in `_COMPLETE`) or an asynchronous event (`T_LISTEN`, `T_CONNECT`, `T_DATA`, `T_EXDATA`, `T_DISCONNECT`, `T_UDERR`, `T_ORDREL`, `T_REQUEST`, `T_REPLY`, `T_PASSCON`). Other events, principally `kOTProviderWillClose` and `T_MEMORYRELEASED` may cause your notifier to be reentered. You can use the functions `OTEnterNotifier` (page 408) and `OTLeaveNotifier` (page 409) to allow your foreground code to limit interruption by notification events.

▲ **WARNING**
For 68000 code it is vital you use the `pascal` keyword when declaring your notifier function. ▲

SPECIAL CONSIDERATIONS

Open Transport can call your notifier function at 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 deferred task time; these restrictions are summarized in the section "Using Notifier Functions to Handle Provider Events" (page 73). Because your notifier can be called at deferred task time, you can only call a limited set of Open Transport functions from your notifier; Table C-3 (page 798) lists the Open Transport functions that you can call at deferred task time.

The following information applies to 68000-based code—whether it runs in emulated mode or on 680x0 machines. Before calling your notifier function, Open Transport restores the A5 register to the value it had when you installed the notifier function. Thus, for applications, your notifier function need not restore its A5 world. But if you're developing a code resource and your development environment references your globals from a register other than A5, your notifier function must save and restore that register.

SPECIAL CONSIDERATIONS

You cannot call any Thread Manager function that will cause the current thread to yield immediately (`YieldToThread` or `YieldToAnyThread`) from within your notification function unless the event code to the notifier is the `kOTSyncIdleEvent` code. Use the `OTUseSyncIdleEvents` function (page 410) to allow synchronous idle events to be sent to your notifier.

SEE ALSO

The `OTRemoveNotifier` function (page 407).

The event codes enumeration (page 383).

Listing 3-1 (page 74).

CHAPTER 22

Endpoints Reference

Contents

CHAPTER 22

This chapter describes the constants, 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 "Providers Reference" (page 383).

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 and the `TBind` structure, are described first, followed by the specific types.

Error-Checking Constant

Open Transport provides a constant that you can use to initialize pointers to the opaque endpoint structure. You can then check that the current endpoint reference is valid before passing it to endpoint functions. The constant is defined as follows:

```
#define     kOTInvalidEndpointRef ((EndpointRef)0)
```

Endpoint Service Types

Open Transport uses the servtype field of the TEndpointInfo structure (page 426) 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 with the OTSnd or OTRcv functions. The constant names that Open Transport can set or return for this parameter are given by the Open Transport flags enumeration.

```
typedef UInt32      OTFlags;
```

```
enum {
    T_MORE              = 0x001,
    T_EXPEDITED         = 0x002,
    T_ACKNOWLEDGED      = 0x004,
    T_PARTIALDATA       = 0x008,
    T_NORECEIPT         = 0x010/,
    T_TIMEDOUT          = 0x020
};
```

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` On sends, the data is sent as expedited data if the endpoint supports expedited data. On receives, the flag indicates that expedited data was sent.

`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 acknowledgment 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.

Open Flags

The following constant is currently reserved. Set to 0 always.

```
typedef UInt32      OTOpenFlags;
```

Endpoint Flags

Open Transport uses the flags field of the TEndpointInfo structure (page 426) 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 (conforms to XTI in XPG4).

T_CAN_SUPPORT_MDATA — This endpoint supports M_DATA, that is, it permits receiving and returning raw packets. For additional information, see “Advanced Topics.”

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). At this time you cannot access this data.

Endpoint States

The OTGetEndpointState function (page 448) 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” (page 89).

```
enum {
        T_UNINIT        = 0,
        T_UNBND         = 1,
        T_IDLE          = 2,
        T_OUTCON        = 3,
        T_INCON         = 4,
        T_DATAXFER      = 5,
        T_OUTREL        = 6,
        T_INREL         = 7
};
```

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 456) allocates a data structure that you specify using one of the constant names given by the structure types enumeration:

```
typedef UInt32 OTStructType

enum {
        T_BIND                = 1,
        T_OPTMGMT             = 2,
        T_CALL                = 3,
        T_DIS                 = 4,
        T_UNITDATA            = 5,
        T_UDERROR             = 6,
        T_INFO                = 7,
        T_REPLYDATA           = 8,
        T_REQUESTDATA         = 9,
        T_UNITREQUEST         = 10,
        T_UNITREPLY           = 11
};
```

Constant descriptions

T_BIND	Specifies the TBind structure (page 429).
T_OPTMGMT	Specifies the TOptMgmt structure (page 574).
T_CALL	Specifies the TCall structure (page 433).
T_DIS	Specifies the TDiscon structure (page 515).
T_UNITDATA	Specifies the TUnitData structure (page 430).
T_UDERROR	Specifies the TUDError structure (page 430).
T_INFO	Specifies the TEndpointInfo structure (page 426).
T_REPLYDATA	Specifies the TReply structure (page 434).
T_REQUESTDATA	Specifies the TRequest structure (page 434).
T_UNITREQUEST	Specifies the TUnitRequest structure (page 431).
T_UNITREPLY	Specifies the TUnitReply structure (page 432).

The TEndpointInfo Structure

The TEndpointInfo structure, which is returned by the OTGetEndpointInfo function, describes the initial characteristics of an endpoint that you opened by calling the OTOpenEndpoint function (page 447) or the OTAsyncOpenEndpoint function (page 441). These functions initialize the supplied TEndpointInfo structure, if you have allocated space for one. You can also get a copy to the TEndpointInfo structure by calling the OTGetEndpointInfo function (page 447).

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 values into your application that you could otherwise get from the endpoint's `TEndpointInfo` field. These values might change. ▲

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

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

Field descriptions

`addr` A value greater than or equal to 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 (e.g. serial communication). However, it does not account for protocols where the address is allowed to be a string, such as a DNS name for TCP/IP or an NBP name for AppleTalk. If you are using this feature, you have to be prepared to allow for a larger address size.

`options` A value greater than or equal to 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` (-2) indicates that this endpoint has no protocol-specific options that you can set; they are read-only. A value of -3 specifies that the provider does not support any options.

`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` (-1) 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 cannot send data with 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 must not send expedited data with 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 or equal to 0 indicates the maximum amount of data (in bytes) that you can send with the `OTConnect` function (page 484) or the `OTAccept` function (page 491). 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 or equal to 0 indicates the maximum amount of data (in bytes) that you can send using the `OTSndDisconnect` function (page 513). A value of `T_INVALID` indicates that this endpoint does not let you send data with disconnection requests. 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 422).

`flags`
A bit field that provides additional information about the endpoint. Possible values are given by the endpoint flags enumeration (page 424).

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 441), the `OTGetProtAddress` function (page 451), and the `OTResolveAddress` function (page 453).

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 441). For connectionless endpoints, this field has no meaning.

The TUnitData Structure

You use the TUnitData structure to describe the data being sent with the OTSndUData function (page 462) and the data being read with the OTRcvUData function (page 467); you pass this structure as a parameter to each of these functions. When sending data you must initialize the buf and len fields of all the TNetbufs. When receiving data, you must initialize the buf and maxlen fields of all the TNetbufs.

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 for address information.

opt
A TNetbuf structure for option information.

udata
A TNetbuf structure for data.

The TUDErr Structure

The TUDErr structure points to information that explains why the OTSndUData function (page 462) has failed. You pass this structure as a parameter to the OTRcvUDErr function (page 465).

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 the buffer referenced by 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. If you are not interested in address information, set `addr.maxlen` to 0.
`opt`	A `TNetbuf` structure that contains information about the options associated with the data sent using the `OTSndUData` function. The `OTRcvUDErr` function fills in the buffer referenced by 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.maxlen` field to 0.
`error`	On return, this 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 469) and the data being read with the `OTRcvURequest` function (page 472); you pass a pointer to this structure as a parameter to each of these functions. When sending data you must initialize the `buf` and `len` fields of all the `TNetbufs`. When receiving data, you must initialize the `buf` and `maxlen` fields of all the `TNetbufs`.

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 for address information.

`opt` A `TNetbuf` structure for option information associated with this request.

`udata` A `TNetbuf` structure for request data.

`sequence` A 32-bit value that specifies the transaction ID for this transaction.

The TUnitReply Structure

You use the `TUnitReply` structure to specify the data being sent with the `OTSndUReply` function (page 475) and the data being read with the `OTRcvUReply` function (page 478). You pass a pointer to the `TUnitReply` structure as a parameter to each of these functions. When sending data you must initialize the `buf` and `len` fields of all the `TNetbufs`. When receiving data, you must initialize the `buf` and `maxlen` fields of all the `TNetbufs`.

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

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

Field descriptions

`addr` A `TNetbuf` structure for address information.

`opt` A `TNetbuf` structure for option information associated with this reply.

`udata` A `TNetbuf` structure for reply data.

sequence — A 32-bit value that specifies the transaction ID for this transaction.

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 484), the OTRcvConnect function (page 487), the OTListen function (page 489), and the OTAccept function (page 491).

If you are using the TCall structure to send information, you must allocate a buffer and initialize it to contain the information. Set the .buf field of each TNetbuf to point to the buffer, and then specify the size of the buffer using the .len field. Set this field to 0 if you are not sending data.

If you are using the TCall structure to receive information, you must allocate a buffer into which the function can place the information when it returns. Then set the .buf field of all the TNetbufs to point to this buffer, and set the .maxlen field to the maximum size of the information. Set this field to 0 if you are not interested in receiving information.

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 an address buffer.

opt — A TNetbuf structure that specifies the location and size of an options buffer.

udata — A TNetbuf structure that specifies the location and size of a buffer for data associated with a connection or disconnection request.

sequence — A 32-bit value used by the OTListen and OTAccept functions to specify the connection ID.

The TRequest Structure

You use the TRequest structure to specify the data being sent with the OTSndRequest function (page 499) and the data being read with the OTRcvRequest function (page 502). You pass a pointer to this structure as a parameter to each of these functions. When sending data you must initialize the buf and len fields of all the TNetbufs. When receiving data, you must initialize the buf and maxlen fields of all the TNetbufs.

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.

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

sequence — A 32-bit value that specifies the transaction ID of the current transaction.

The TReply Structure

You use the TReply structure to specify the data being sent with the OTSndReply function (page 504) and the data being read with the OTRcvReply function (page 507). You pass this structure as a parameter to each of these functions. When sending data you must initialize the buf and len fields of all the TNetbufs. When receiving data, you must initialize the buf and maxlen fields of all the TNetbufs.

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.

opt
: A TNetbuf structure describing the location and size of the options buffer.

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

The TDiscon Structure

You use the TDiscon structure to specify data sent with OTSndDisconnect (page 513) and retrieved by the OTRcvDisconnect function (page 515). You pass this structure as a parameter to these functions.

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 references data sent with the OTSndDisconnect function or received by the OTRcvDisconnect function.

`reason` A 32-bit value 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 32-bit value 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 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 Type of Service" (page 85).

You can also use general provider functions with endpoints. General provider functions and structures are described in "Providers Reference" (page 383).

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 the endpoint.

For more information about binding an endpoint, see "Binding and Unbinding Endpoints" (page 441). For a description of the `OTCloseProvider` function, see "Providers Reference" (page 383).

OTOpenEndpoint

Opens an endpoint. This function is synchronous, and creates an endpoint that operates synchronously.

C INTERFACE

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

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(page 376). The `OTOpenEndpoint` function disposes of the configuration structure before returning.
`oflag`	Reserved; must be set to 0.
`info`	A pointer to a `TEndpointInfo` structure (page 426) 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.
function result	An endpoint reference for the endpoint that has been created or `NULL` if the endpoint has not been created.

DISCUSSION

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 kOTInvalidEndpointRef.

The function creates an endpoint that is synchronous, does not block, and does not acknowledge sends. To change these settings, you can call the OTSetAsynchronous function (page 395), the OTSetBlocking function (page 398) and the OTAckSends function (page 401).

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 (page 441).

To close and delete an endpoint, call the OTCloseProvider function (page 392).

SPECIAL CONSIDERATIONS

The OTOpenEndpoint function can only be called at system task time because it is a synchronous function.

The OTOpenEndpoint function disposes of the configuration structure passed in as a parameter. If you want to use the same configuration to open additional endpoints, you must obtain a copy of the configuration structure before calling the OTOpenEndpoint function, by calling the OTCloneConfiguration function (page 378).

SEE ALSO

"Creating a Configuration Structure" (page 35).

"Reusing Provider Configurations" (page 37).

The OTAsyncOpenEndpoint function (page 438).

"Endpoint States" (page 89).

OTAsyncOpenEndpoint

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

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. The `OTAsyncOpenEndpoint` function disposes of the configuration structure before returning.

`oflag` Reserved; must be set to 0.

`info` A pointer to a `TEndpointInfo` structure (page 426) 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. Never pass a universal procedure pointer as the `proc` parameter.

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

function result An error code. See Discussion.

DISCUSSION

The OTAsyncOpenEndpoint function opens an endpoint having the characteristics specified by the config parameter. How processing proceeds 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 has queued the operation for execution. When it completes it calls the notifier function that you specified in the proc parameter, passing the event 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. For information about notifier functions, see “Application-Defined Notifier Functions” (page 413) and “Event Codes”(page 451).

An endpoint created by the OTAsyncOpenEndpoint function operates in asynchronous mode, does not block, and does not acknowledge sends. To change these default settings, you can call the OTSetSynchronous function (page 394), OTSetBlocking function(page 398) and the OTAckSends function (page 401).

The initial 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 (page 441).

SPECIAL CONSIDERATIONS

The first time you open a provider of any kind, Open Transport allocates needed resources at system task time, calling the ScheduleSystemTask function to do this. To allow this processing to take place, you need to call SystemTask or WaitNextEvent from a foreground task while waiting for the endpoint to be created.

The OTAsyncOpenEndpoint function disposes of the configuration structure returned by the OTCreateConfiguration function(page 376). 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(page 378) before you call the OTAsyncOpenEndpoint function. For more information, see “Configuring and Opening a Provider” (page 34).

SEE ALSO

The `OTOpenEndpoint` function (page 437).

Table 4-4 (page 95).

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.

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.

OTBind

Assigns an address to an endpoint.

C INTERFACE

```
OSStatus OTBind(EndpointRef ref,
                TBind* reqAddr,
                TBind* retAddr);
```

C++ INTERFACE

```
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 429) 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 `NULL` for the `reqAddr` parameter, Open Transport chooses a protocol address for you and requests 0 as the endpoint's maximum number of outstanding connect indications.

If you want Open Transport to assign an address for you, while still specifying a `qlen` value, set the `addr.len` field of the `TBind` structure to 0.

`retAddr` A pointer to a `TBind` structure (page 429) that, on return, indicates the address to which the endpoint is actually bound and, for connection-oriented endpoints, indicates the maximum number of outstanding connect indications that this endpoint actually allows. The `TBind` structure is described on (page 429).

You can set this parameter to `NULL` if you do not care to know what address the endpoint is bound to or what the negotiated value of `qlen` is. However, it is almost always a good idea to check this field to find out whether the endpoint negotiated a different queue length from the length you requested.

function result An error code. See Discussion.

DISCUSSION

You call the `OTBind` function to request that an endpoint be bound to an address. 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 `NULL` 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.

Before you call the function, you must allocate a buffer for the address and set the retAddr->addr.buf field to point to it. You must also specify the maximum size of the address using the retAddr->addr.maxlen field. 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. The field retAddr->addr.len contains the length of the address. If the ret->addr.maxlen field indicates a buffer size that is not large enough to contain the address, the function returns with the result kOTBufferOverflowErr. If the requested address is not available, the function returns the result kOTAddressBusyErr. If the endpoint cannot allocate an address, the function returns with the kOTNoAddressErr result.

If the endpoint is in synchronous mode, the function returns when the bind is complete. If the endpoint is in asynchronous mode and you have installed a notification routine, the OTBind function returns kOTNoError and sends the T_BINDCOMPLETE event to your notifier when the bind completes. For more information on notifier functions and event codes, see MyNotifierCallback function(page 413) and "Event Codes" (page 383). If 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 (page 448). 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 (page 445).

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. You can set this field to 0 if you are only going to be initiating outgoing connections. 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.

You must not bind more than one connectionless endpoint to a single address. If you attempt to bind a second endpoint to the an address that is already bound, the OTBind function will return the result kOTAddressBusyErr. 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 on 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.

SEE ALSO

The `OTCloseProvider` function (page 392).

Table 4-4 (page 95).

"Processing Multiple Connection Requests" (page 111).

The `OTConnect` function (page 484).

The `OTAccept` function (page 491).

"TCP/IP Services" (page 237).

"AppleTalk Data Stream Protocol (ADSP)" (page 319).

"Datagram Delivery Protocol (DDP)" (page 310).

"Printer Access Protocol (PAP)" (page 343).

OTUnbind

Dissociates an endpoint from its address or cancels an asynchronous call to the `OTBind` function.

C INTERFACE

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

C++ INTERFACE

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

PARAMETERS

`ref`	The endpoint reference of the endpoint that you are unbinding.
function result	An error code. See Discussion.

DISCUSSION

If the endpoint is in synchronous mode, the function returns when the unbind is completed.

If the endpoint is in asynchronous mode and you have installed a notifier, the function returns `kOTNoError` and the provider sends the event code `T_UNBINDCOMPLETE` to your notifier when the unbind is completed. For more information on notifier functions and event codes, see `MyNotifierCallback` function (page 413) and "Event Codes" (page 383). If you have not installed a notifier function, the only way to determine that the endpoint has been unbound is to use the `OTGetEndpointState` function (page 448) to poll the state of the endpoint. The function returns the `kOTStateChangeErr` result until the unbind completes. If the function succeeds, the state of the endpoint is `T_UNBND`. If it fails, its state is `T_IDLE`.

For connectionless endpoints, a common error code is `kOTLookErr`, usually indicating that more data has arrived. Since XTI defines that the `OTUnbind` function can succeed only when there is no data available, the only recourse is to either read the data and call the `OTUnbind` function again or to close the endpoint.

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.

SEE ALSO

The `OTBind` function (page 441).

The `OTCloseProvider` function (page 392).

Table 4-4 (page 95).

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 type 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 other functions return information about an endpoint's address: the `OTGetProtAddress` function 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 modes of execution (`OTIsSynchronous`, `OTIsAckingSends`, `OTIsNonBlocking`). For more information about these functions, see "Providers Reference" (page 383).

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 426) 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.

function result An error code. See Appendix B(page 785).

DISCUSSION

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

If the endpoint is in synchronous mode, the function fills out the fields in the `TEndpointInfo` structure and returns. If the endpoint is in asynchronous mode and you have installed a notifier function, the `OTGetEndpointInfo` function returns the result `kOTNoError` and sends the `T_GETINFOCOMPLETE` event to your notifier when the operation completes. The `result` parameter is `kOTNOError` if the function succeeded. Otherwise, it contains a result code describing the reason why it failed. The `cookie` parameter passed to the notification routine contains the value of the `info` parameter that you originally passed to the `OTGetEndpointInfo` function. If you have not installed a notification routine, it is not possible to determine when this command completed. For more information on notification functions and event codes, see `MyNotifierCallback` function (page 413) and “Event Codes”(page 383).

SEE ALSO

The `OTGetEndpointState` function (page 448).

“The TEndpointInfo Structure” (page 426)

Table 4-4 (page 95).

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.
function result	A non-negative number indicating the state of the endpoint. Or, a negative number if `ref` is invalid.

DISCUSSION

The `OTGetEndpointState` function returns an integer greater than or equal to 0 indicating the state of the specified endpoint. The endpoint state enumeration (page 424) describes possible endpoint states and lists their decimal value.

This function returns endpoint state information immediately, whether the endpoint is in synchronous or asynchronous mode.

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.

The only time this function fails is if you have provided an invalid endpoint reference.

SEE ALSO

The `OTGetEndpointInfo` function (page 447).

"Endpoint States" (page 89) and "Endpoint States" (page 424).

Table 4-4 (page 95).

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.
function result	A positive value specifying the pending asynchronous event, or a negative number corresponding to an error result code. For a list of result codes, see Appendix B(page 785).

DISCUSSION

This function returns the current pending asynchronous event on the endpoint. The function returns immediately, regardless of whether the endpoint is in synchronous or asynchronous mode.

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 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 4-8 (page 105) lists the functions that might return the `kOTLookErr` result and the events that can cause these functions to fail.

IMPORTANT

The `OTLook` function can actually clear certain asynchronous events, specifically the `T_GODATA` and the `T_GOEXDATA` events. If you consume one of these events by calling `OTLook`, you will not be notified of that type of event until you are flow controlled again. ▲

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, rather than using the `OTLook` function.

SEE ALSO

"Handling Events for Endpoints" (page 102).

"Event Codes" (page 383).

"Application-Defined Notifier Functions" (page 413).

Table 4-4 (page 95).

OTGetProtAddress

Obtains the address to which an endpoint is bound and, if the endpoint is currently connected, also 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 429). The `boundAddr->addr` field is a `TNetBuf` structure that returns the address of the endpoint specified by the `ref` parameter. You must allocate a buffer for the address information and initialize the `boundAddr.buf` field to point to that buffer. You must also initialize the `boundAddr.maxlen` field to the size of the address buffer.

If the endpoint is in the `T_UNBND` state, the field `boundAddr->addr.len` field is set to 0.

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 429). If the peer endpoint is currently connected or is in the `T_DATAXFER` state, the `peerAddr->addr` field (a `TNetbuf` structure) returns the address of the endpoint's peer.

If you are calling this function only to determine the address to which the endpoint is bound, you can set the `pperAddr` parameter to `nil`.

The `peerAddr->qlen` field is ignored. If the endpoint is not connected or in the `T_DATAXFER` state, the `peerAddr->addr.len` field is set to 0 and the `peerAddr->addr.buf` pointer may be set to `NULL`.

function result An error code. See Discussion.

DISCUSSION

The `OTGetProtAddress` function returns the address to which an endpoint is bound in the `boundAddr` parameter and, if the endpoint is currently connected, it returns the address of its peer in the `peerAddr` parameter. Not all endpoints support this function. If the `T_XPG4_1` bit in the `flags` field of the `TEndpointInfo` structure (page 424) is not set, the endpoint does not support this function.

You are responsible for allocating 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 as the size of the buffers.

If the endpoint is in synchronous mode, the function returns when the operation is complete. If the endpoint is in asynchronous mode and a notification routine is installed, the function returns `kOTNoError` and the provider sends the event code T_GETPROTADDRCOMPLETE, when the operation completes. The result parameter is kOTNoError if the function succeeded or a negative error code if it did not. The cookie parameter sent to the notification routine contains the peerAddr value unless that is NULL. If the peerAddr value was NULL, the cookie parameter contains the boundAddr value instead. If a notifier is not installed, it is not possible to determine when the function completes. For more information on notifier codes and event codes, see MyNotifierCallback function (page 413) and “Event Codes”(page 383).

SEE ALSO

The OTAccept function (page 491).

Table 4-4 (page 95).

OTResolveAddress

Returns the protocol address that corresponds to a high-level address on an endpoint.

C INTERFACE

```
OSStatus OTResolveAddress(EndpointRef ref,
                        TBind* reqAddr,
                        TBind* retAddr
                        OTTimeout timeOut);
```

C++ INTERFACE

```
OSStatus TEndpoint::ResolveAddress(TBind* reqAddr, TBind* retAddr);
```

PARAMETERS

ref The endpoint reference whose provider should do the address resolution.

`req` A pointer to a `TBind` structure (page 429). The field `reqAddr->addr.buf` points to a buffer containing the high-level address. This address must be in an appropriate format for the protocol family. For example, for AppleTalk this must be an `NBPAddress`.

`ret` A pointer to a `TBind` structure (page 429). The `retAddr->addr.buf` field points to a buffer that you filled out with the lowest-level address that corresponds to the address referenced by the `reqAddr->addr.buf` field of the `reqAddr` parameter.

`timeout` The maximum time in milliseconds that you want to wait for address resolution. Not all protocols honor this requirement.

function result An error code. See Discussion.

DISCUSSION

The `OTResolveAddress` function returns the lowest-level address for your endpoint. Not all endpoints support this function. If the `CAN_RESOLVE_ADDR` bit in the `flags` field (page 424) of the `TEndpointInfo` structure is set, the endpoint supports this function. Using this function saves you the trouble of opening and closing a mapper if the only reason you have for opening the mapper is to look up the address corresponding to a specific endpoint name. This function is also useful in that you don’t need to know the underlying protocol to resolve an address.

You are responsible for allocating the buffers described by the `reqAddr` and `retAddr` parameters required to hold the addresses. To determine how large these buffers should be, examine the `addr` field of the `TEndpointInfo` structure for the endpoint (page 426), which specifies the maximum amount of memory needed to store an address for the endpoint specified by the `ref` parameter.

If the endpoint is in synchronous mode, the funciton returns when the operation is complete. If the endpoint is in asynchronous mode and you have installed a notification routine, the `OTResolveAddress` function returns immediately with the `kOTNoError` result and sends the `T_RESOLVEADDRCOMPLETE` event code to your notifier when the operation completes. See Appendix B(page 785). The `result` parameter is `kOTNoError` if the function succeeded or a negative result code if it did not. The `cookie` parameter contains the `ret` parameter. If a notifier is not installed, it is not possible to determine when the `OTResolveAddress` function completes. For more information on notifier functions and event codes see `MyNotifierCallback` function(page 413) and "Event Codes" (page 383).

OTSync

Ensures that the endpoint provider and the Open Transport client libraries 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.
function result	See Discussion.

DISCUSSION

The provider's and the client's libraries knowledge about an endpoint's state might get out of sync if the provider and the client libraries occupy different memory spaces. The current run-time environment does not support separate memory spaces; therefore, this function does nothing and is 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.

If the `OTSync` function fails, it returns a negative integer corresponding to a result code. See Appendix C(page 793).

If the endpoint is in synchronous mode, it returns as soon the operation is complete. If the endpoint is in asynchronous mode and you have installed a notification routine, the OTSync function returns the result kOTNoError and the provider sends the T_SYNCCOMPLETE event code to the notifier when the operation completes. The result parameter is either a negative error code or a state code. The cookie parameter has no meaning. If a notifier is not installed, it is not possible to determine when the OTSync function completes. For more information on notifier functions and event codes see MyNotifierCallback function(page 413) and "Event Codes" (page 383).

SEE ALSO

The OTGetEndpointState function (page 448).

"Endpoint States" (page 424).

Table 4-4 (page 95).

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 "Providers" (page 61).

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,
                       UInt32 fields,
                       OSStatus* err = NULL);
```

PARAMETERS

ref — The endpoint reference of the endpoint for which the data structure is allocated.

structType — A 32-bit value 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 425).

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. See Discussion for more information.

You can specify one or more constant names for the fields parameter. See Discussion for the value and meaning of these names. To specify more than one constant name, use the bitwise OR operator to combine values.

err — A result code. See Appendix B (page 785) for more information.

DISCUSSION

The OTAlloc function allocates a data structure for use in a subsequent endpoint call. You use the structType parameter to specify the structure to be allocated and the fields parameter to specify the substructures to be allocated. Possible field values are given by the following enumeration

```
enum
{
    T_ADDR      = 0x01,     /* for address information */
    T_OPT       = 0x02,     /* for option information */
    T_UDATA     = 0x04,     /* for data */
    T_ALL       = 0xffff    /* for address, option, and data */
};
```

If the OTAlloc function succeeds, it returns a pointer to the desired structure.

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 bufferis. 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.

If the requested structure contains TNetbuf fields and you do not specify these fields using the fields parameter, the OTAlloc function sets the maxlen and len fields to 0 and the buf field to NIL.

Note
If you just want to allocate a block of memory, consider using the OTAllocMem function (page 625). ◆

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.

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.

You must not use the pointer returned by the OTAlloc function in calls to any other endpoint as other endpoints mayhave different sizes.

SEE ALSO

The OTFree function (page 459).

The TBind structure (page 429).

The TEndpointInfo structure (page 426).

The OTAllocMem function (page 625) and the OTFreeMem function (page 625).

Table 4-4 (page 95).

OTFree

Frees memory allocated using the OTAlloc function.

C INTERFACE

```
OSResult 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 425).

function result An error code. See Appendix B(page 785).

DISCUSSION

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. The OTAlloc function (page 456) allocates the memory OTFree deallocates.

▲ **WARNING**
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. If you have changed these, you must restore them to their original value before calling OTFree. ▲

SPECIAL CONSIDERATIONS

Although this function is defined to return an OTResult, there are no meaningful positive results. You either get kOTNoError, or a negative error code.

SEE ALSO

The OTAlloc function (page 456).

The OTAllocMem function (page 625) and the OTFreeMem function (page 625).

Table 4-4 (page 95).

Determining if Bytes Are Available

Open Transport provides the function `OTCountDataBytes` to determine whether bytes are available to be read from an endpoint (without actually reading them).

OTCountDataBytes

Returns the amount of data currently available to be read.

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 whose pending data you wish to count.

`countPtr` A pointer to a value that is set to the size (in bytes) of the data in the first packetwaiting to be read from the endpoint.

function result See Appendix B(page 785).

DISCUSSION

If the function returns successfully, the `countPtr` parameter points to a buffer containing an approximation of the the number of bytes in the message buffer at the top of the stream.

What the function counts depends on the type of endpoint. If it is packet-oriented, the function counts the number of bytes in the first packet. If it's stream-oriented and 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 first part.

You can call this function upon receipt of a T_DATA event to get an approximation of 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. But, you should be careful not to assume that this is all the data. You should always read data until you get the kOTNoDataErr result.

Because what this function counts depends on which event is the most current outstanding event and because other events can occur before the function can complete, never use this count as more than a hint.

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, or IP is described at greater length in the section "Using Connectionless Transactionless Service" (page 119). This section describes the functions used to send and receive data, OTSndUData and OTRcvUData. You use the TUnitData structure (page 430) with these functions to specify the data being transferred.

Due to the nature of connectionless transactionless service, you are not notified if the data fails to reach its destination. 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 449) 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.

SEE ALSO

Table 4-4 (page 95).

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 430) that specifies the data to be sent, its destination, and any options for this send. The `udata->addr.buf` field is a pointer to a buffer that contains the address to which the data is sent. You must initialize the buffer, and set the `udata->addr.len` field to the size of the address. The `udata->opt.buf` field is a pointer to a buffer that contains option information. Store option information in the buffer and set the `udata->opt.len` field to the size of the options. If you do not want to specify any options, set this field to 0. The `udata->udata.buf` field is a pointer to a buffer that contains data to be sent. Set the `udata->udata.len` field to the size of the data.
function result	An error code. See Discussion.

DISCUSSION

The `OTSndUData` function sends data to the address specified by the `udata->addr` field of the `udata` parameter. If the endpoint does not support sending 0 bytes or if the amount of data exceeds the TSDU size for the endpoint, the function returns with the result `kOTBadDataErr`.

If the endpoint is in synchronous blocking mode, the `OTSndUData` function returns when the send is complete. 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 or asynchronous 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 sends your notifier function, a `T_GODATA` event. 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 (page 449).

The next table shows how the endpoint's modes of operation 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 might be returned.	The `kOTFlowErr` result might be returned.

SPECIAL CONSIDERATIONS

Some endpoint providers are not able to detect immediately whether you specified incorrect address or option information. Because of this, the provider calls your notifier function when it detects the error, passing the `T_UDERR` event code 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 (page 465) 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 a state where it cannot do other sends, nor deallocate memory reserved for internal buffers associated with the send.

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. See "Option Management" (page 165) for information on setting this option.

SEE ALSO

The OTRcvUData function (page 467).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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 430) that specifies, on return, the address of the endpoint to which data was sent, the options specified for the send, and the reason for the error.

The uderr->addr.buf field is a pointer to a buffer that contains the destination address of the data sent. You must allocate this buffer and set the uderr->addr.maxlen field to the maximum size for the buffer.

The uderr->opt.buf field is a pointer to a buffer that contains the value of options you specified for the send. You must allocate a buffer for this data and set the uderr->opt.maxlen field to the maximum size of the buffer.

The uderr->error field is a 32-bit value that specifies a protocol-dependent error code for the OTSndUData function that failed.

function result An error code. See Discussion.

DISCUSSION

You use the OTRcvUDErr function if you have called the OTSndUData function (page 462) and the endpoint provider has issued the T_UDERR event to indicate that the send operation did not succeed. This usually happens when 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 NULL. 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 all subsequent functions for this endpoint return the result kOTLookErr.

SEE ALSO

Table 4-4 (page 95).

OTRcvUData

Reads data sent to 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 430) that, on return, contains information about the data that has been received.

The `udata->addr.buf` field is a pointer to a buffer that is filled with the address of the endpoint that has sent the data. You must allocate this buffer and set the `udata->addr.maxlen` field to the size of the buffer.

The `udata->opt.buf` field is a pointer to a buffer that is filled with 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 and you must set the `opt.maxlen` field to the size of the buffer.

The `udata->udata.buf` field is a pointer to a buffer in which the function stores data when it returns. You must allocate a buffer for the data and set the `udata.maxlen` field to the size of the buffer.

`flag` A pointer to an unsigned long variable whose bit setting, on return, indicates whether you need to retrieve more data. If the `T_MORE` is set, there is more data; if it is clear, there is no more data.

function result An error code. See Discussion.

DISCUSSION

The `OTRcvUData` function returns information about the data read into the `TUnitData` structure.

The `OTFlags` variable, referenced by the `flag` parameter, indicates whether there is more data to be retrieved in this packet. If the buffer referenced 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 `flag` 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 part of the unit 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 (page 449) 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 to receive a `T_DATA` event, but when you execute the `OTRcvUData` function, it returns with a `kOTNoDataErr` result. If this happens, you should continue as though you just finished reading all the data.

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. See "Option Management" (page 165) for information on setting this option.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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" (page 123).

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,
                          OTFlags reqFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndURequest(TUnitRequest* req,
                          OTFlags reqFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint making the request.

`req` A pointer to a `TUnitRequest` structure (page 431) that specifies the address of the responder, the request data, and the ID of this transaction.

The `req->addr` 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 `req->addr.buf` field to point to this buffer and set the `req->addr.len` field to the length of the address.

The `req->opt` 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 `req->opt.buf` field to point to it. You must set the `req->opt.len` field to the length of the option data or to 0 if you don't want to specify any options.

The `req->udata` field specifies the location and size of a buffer containing the request data. You must allocate a buffer for the request data, initialize the `req->opudata.buf` field to point to it, and set the `req->opudata.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.

You set the `req->sequence` field to a unique, non -zero value when you send the request. You only need to do this if the endpoint issues multiple requests.

`reqFlags` A bitmapped 32-bit value 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 bitwise OR operator to set both values.

function result An error code. See Discussion.

DISCUSSION

You use the `OTSndURequest` function to initiate a transaction. When the responder replies to your request, you use the `OTRcvUReply` function (page 478) to read the reply. If the endpoint is issuing multiple requests, you should set the `sequence` field of the `req` parameter to a unique non-zero number so that each request is distinguished from all other outstanding requests issued by the endpoint.

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.

The responder's 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 (page 475) 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 (page 426).

If the endpoint is in synchronous 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 asynchronous or 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 (page 449) or by 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 kOTFlowErr result is never returned.	The function returns if flow-control restrictions are in effect or the request data has been accepted by the provider . The kOTFlowErr result might be returned.
Asynchronous	The function returns immediately The kOTFlowErr result might be returned.	The function returns immediately. The kOTFlowErr result might be returned.

SEE ALSO

Table 4-4 (page 95).

"AppleTalk Reference"(page 721)

OTRcvURequest

Reads a request sent by a client using a connectionless transaction-based protocol.

C INTERFACE

```
OSStatus OTRcvURequest(EndpointRef ref,
                       TUnitRequest* req,
                       OTFlags* reqFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvURequest(TUnitRequest* req,
                        OTFlags* reqFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint accepting the request.

`req` A pointer to a `TUnitRequest` structure (page 431) that contains information about the request being received.

The `req->addr` 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 `req->addr.buf` field to point to it. You must also set the `req->addr.maxlen` field to the maximum size of the address.

The `req->opt` 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 `req->opt.buf` field to point to this buffer. You must set the `req->opt.maxlen` field to the maximum size necessary to hold option information for the endpoint.

The `req->udata` 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 `req->udata.buf` field to point to it. You must set the `req->udata.maxlen` field to the maximum size of the request data.

The value of the `req->sequence` field is generated by the endpoint provider when you read the request. You need to save this value and use it for the `req->sequence` field when sending a reply or cancelling the transaction.

`reqFlags` A bitmapped 32-bit value 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 request. It is possible that all flags are set.

function result An error code. See Discussion.

DISCUSSION

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 buffer you allocated for the address is not big enough, the function returns with the kOTBufferOverflowErr result and the incoming request is dropped.

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 (page 449) or use a notifier function to retrieve this event. You should then call the funtion to retrieve request data until the function returns with the kOTNoDataErr result.

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.

SEE ALSO

The OTSndUReply function (page 475).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

OTSndUReply

Replies to a request sent by a client using a connectionless transaction-based protocol.

C INTERFACE

```
OSStatus OTSndUReply(EndpointRef ref,
                     TUnitReply* reply,
                      OTFlags replyFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::SndUReply(TUnitReply* reply,
                     OTFlags replyFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint sending the reply.

`reply` A pointer to a `TUnitReply` structure (page 432) that specifies information about the requester, options, the ID of this transaction, and the reply data.

The `reply->addr` 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 `reply->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 `reply->addr.buf` field to point to the buffer and set the `reply->addr.len` field to the length of the address.

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

The `reply->udata` 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 `reply->udata.buf` field to point to that buffer, and set the `reply->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.

Set the `reply->sequence` field to the value that you read for this field with the `OTRcvURequest` function (page 472).

`replyFlags` A bitmapped 32-bit value, which you can set to `T_MORE` to indicate that you are sending more reply data with a subsequent call to the `OTSndUReply` function.

function result An error code. See Discussion.

DISCUSSION

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, any options you want to specify for this reply, and a transaction ID. 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 you do not receive an acknowledgement from the other side within a specific amount of time (usually negotiated with the ATP_OPT_RELTIMER option), 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 code 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 (page 472). 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).	The function returns immediately.
	The kOTFlowErr result is never returned.	The kOTFlowErr result might be returned.

	Blocking	Nonblocking
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 might be returned.	The `kOTFlowErr` result might be returned.

SEE ALSO

The `OTCancelUReply` function (page 483).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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* replyFlags);
```

C++ INTERFACE

```
OSStatus TEndpoint::RcvUReply(TUnitReply* reply,
                       OTFlags* replyFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint accepting the reply.

`reply` A pointer to a `TUnitReply` structure (page 432) that specifies the location to store the reply information.

The `reply->addr` 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 `reply->addr.buf` field to point to this buffer. You must also set the `reply->addr.maxlen` field to the maximum size of the buffer.

The `reply->opt` 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->opt.maxlen` field to the maximum size necessary to hold option information.

The `reply->udata` 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 `reply->udata.buf` field to point to it, and set the `reply->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 have sent out multiple requests, you can use the `reply->sequence` field to match incoming replies to outgoing requests.

`replyFlags` 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.

function result An error code. See Discussion.

DISCUSSION

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. If the OTRcvUReply function returns withthe T_PARTIALDATA flag set, 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 will 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 read with the OTRcvUReply function does not constitute the entire reply; more data is coming but it has not yet arrived. 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 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.

The following table shows how the endpoint's mode of execution and blocking status affects the behavior of the OTRcvUReply function.

	Blocking	**Nonblocking**
Synchronous	The function returns when the provider lifts flow-control restrictions and the reply has arrived.	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 provider calls your notifier, passing T_REPLY for the code parameter.	The provider calls your notifier, passing T_REPLY for the code parameter.
	The kOTFlowErr result might be returned.	The kOTFlowErr result might be returned.

SEE ALSO

The OTSndURequest function (page 469).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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 32-bit value 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 for the endpoint. If you specify an invalid sequence number, Open Transport does not do anything.

function result If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment.

DISCUSSION

The `OTCancelURequest` function cancels the outgoing request whose transaction ID is specified by the `seq` parameter.

When you call the `OTSndURequest` function (page 469), 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.

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` (page 483) to cancel an *incoming* request.

SEE ALSO

Table 4-4 (page 95).

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 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 for the endpoint. If you specify an invalid sequence number, Open Transport does not do anything.

function result If the function completes successfully, it returns the `kOTNoErr` result; it does not return any other kind of acknowledgment.

DISCUSSION

The `OTCancelUReply` function cancels the incoming request whose transaction ID is specified by the `seq` parameter.

When you call the `OTRcvURequest` function (page 472), 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.

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 (page 481) to cancel an *outgoing* request.

Establishing Connection

To use a connection-oriented endpoint you must first bind the endpoint, then you must use the functions described in this section to establish the connection. The active peer uses the `OTConnect` and `OTRcvConnect` functions; the passive peer 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" (page 512).

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 433). You must allocate buffers, store the appropriate address, option, and data in them, and specify their length.

The `sndCall->addr.buf` field points to a buffer that specifies the address of the passive peer. The `sndCall->addr.len` field specifies the size of the address in bytes.

The `sndCall->opt.buf` field points to a buffer that specifies the options you want to negotiate. The `sndCall->opt.len` field specifies the size of the buffer in bytes.

The `sndCall->udata.buf` field points to a buffer that contains data you want to send with the connection request. The `sndCall->udata.len` field specifies the size of the data in bytes.

`rcvCall` A pointer to a `TCall` structure (page 433) that stores information about the connection when the function returns. You can set this parameter to nil, inwhich case no information is returned to you.

This parameter is only meaningful for synchronous calls to the `OTConnect` function. In this case, you must allocate the buffers for address, options, and data returned, and you must specify their maximum length. You can set this to nil if you'e not interested in the result.

The `rcvCall->addr.buf` field points to a buffer that is used to store the address to which you are actually connected. Set the `rcvCall->addr.maxlen` field to the size of the buffer in bytes.

The `rcvCall->opt.buf` field points to a buffer in which the options that have actually been negotiated are stored. Set the `rcvCall->opt.maxlen` field to the size of the buffer in bytes.

The `rcvCall->udata.buf` field points to a buffer that contains data sent by the peer accepting the connection. Set the `rcvCall->udata.maxlen` field to the size of the buffer in bytes.

function result An error code. See Discussion.

DISCUSSION

If the endpoint is in synchronous mode, the `OTConnect` function returns after the connection is established and fills in the fields of 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 (page 449) 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 the function `OTRcvConnect` (page 487). If the event is `T_DISCONNECT`, you must call the `OTRcvDisconnect` function (page 515) 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 in the `rcvCall` parameter if the endpoint were in synchronous mode.

It is possible that the remote address returned in the `addr.buf` field of the `rcvCall` parameter is not the same as the address you requested using the `sndCall->addr` field. This happens when the remote peer accepts the connection on 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 (page 426) for the endpoint to determine if the endpoint supports the sending of data and to determine the maximum size of the data.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

OTRcvConnect

Reads the status of an 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 433) that, on return, contains information about the newly established connection. You can set this parameter to `nil`, in which case no information is returned to you. Otherwise, you must allocate buffers for the address, option, and data returned, and you must specify the maximum length for each buffer.

The `call->addr.buf` field points to a buffer that specifies the address to which you are actually connected. Set the `call->addr.maxlen` field to the size of the address.

The `call->opt.buf` field points to a buffer that contains the option values that were negotiated for this connection. Set the `call->opt.maxlen` field to the size of the buffer.

The `call->udata.buf` field points to a buffer that contains any data sent as part of the conection request. Set the field `call->udata.maxlen` to the size of the buffer.

The `call->sequence` field is not used by this function.

function result An error code. See Discussion.

DISCUSSION

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. You can examine the `connect` field of the `TEndpointInfo` structure (page 426) to determine whether your endpoint supports the receiving of data with a connection request.

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 (page 449) to verify that a `T_DISCONNECT` event is the reason for the `kOTLookErr`, and then you should call the `OTRcvDisconnect` function (page 515) 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, or it returns the result `kOTNoError` and sets up the call structure appropriately.

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.

SEE ALSO

The `OTConnect` function (page 484).

“AppleTalk Reference” (page 721).

Table 4-4 (page 95).

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 433) that contains, on return, 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 must allocate buffers in which this information can be stored and specify the maximum length for each buffer. The `call->addr.buf` field points to a buffer that will hold the address of the endpoint that requested the connection. Set the `call->addr.maxlen` field to the size of the buffer. The `call->opt.buf` field points to a buffer that will hold the options that the peer has requested for this connection. Set the `call->opt.maxlen` field to the size of the buffer.

The `call->udata.buf` field points to a buffer that stores any data sent by the endpoint requesting the connection. Set the `call->udata.maxlen` field to the maximum size of this buffer.

The `call->sequence` field contains the connection ID of the incoming request.

function result An error code. See Discussion.

DISCUSSION

You use the `OTListen` function to listen for incoming connection requests.

If the endpoint is in synchronous mode and is blocking, the `OTListen` function returns when a connection request has arrived. 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 (page 513), or you can accept the request using the `OTAccept` function (page 491).

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. Typically, you would call the `OTListen` function from within a notifier function in response to the `T_LISTEN` event. .

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.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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 433) 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.

The `call->addr.buf` field points to a buffer that contains the address of the peer that requested the connection. The provider may optionally check this address to ensure you're accepting the right connection. If you do not want to specify a value, set the `call->addr.len` field to 0.

The `call->opt.buf` field points to a buffer that contains option values for this connection. If you do not want to specify any options set the `call->opt.len` field to 0.

The `call->udata.buf` field points to a buffer containing any data to be returned to the endpoint requesting the connection. The `call->udata.len` field specifies the length of the data.

The call->sequence field specifies 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.

function result An error code. See Discussion.

DISCUSSION

You use the OTAccept function to accept a request that you retrieved using the OTListen function (page 489). You can accept a connection on either the same or 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.

If you do not wish to accept the connection request, you must call the OTSndDisconnect function (page 513).

If the endpoint is in synchronous mode, the function does not return until the operation is complete. 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 the event T_ACCEPTCOMPLETE for the code parameter and the endpoint reference (ref) 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 resRef 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.

Note
It is possible, in the case where the listening and accepting endpoints are different, that the accepting endpoint receives a T_DATA event before receiving the T_PASSCON event. If this happens, set a flag to defer receiving the data until later. When the T_PASSCON event is received, check the flag and issue the OTRcv call if the flag is set. Note that after deferring the handling of the T_DATA event, your handler will not be notified of this event again, until you read all of the data presently available. ◆

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 result kOTNoError.

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.

Calling the OTAccept function on an endpoint that was bound with a qlen greater than 1 can result in the result kOTLookErr being returned because another T_LISTEN event or T_DISCONNECT event has arrived. For information on how to handle this situation, see "Handling Multiple Simultaneous Connections" (page 142).

SEE ALSO

The OTBind function (page 441).

"TCP/IP Services" (page 237)

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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. Similarly, 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 using a connection-oriented, transactionless endpoint.

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.

nbytes	A 32-bit value specifying the number of bytes being sent.
flags	A 32-bit bitmapped value 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 bitwise OR operator.
function result	A positive number indicating the actual number of bytes sent or a negative number indicating an error code. See Discussion.

DISCUSSION

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.

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 (page 426) for the endpoint or by the etsdu field for expedited data (T_EXPEDITED flag set).

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 set the T_EXPEDITED bit of the flags parameter.

If you want to break up the data into 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, turn off the T_MORE flag. Only some endpoints support the preservation of logical units; you should check the tsdu field of the TEndpointInfo structure to find out if yours does.

If the endpoint is in synchronous 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 non-blocking or asynchronous 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 sendsa T_GODATA event to let you know that it is possible to send data. You should wait for this event and then send the data again.

If the endpoint is in non-blocking or asynchronous mode, it is also 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. In this case, you should call the function again to send the remaining data.

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 modes of operation affects the behavior of the OTSnd function.

	Blocking	**Nonblocking**
Synchronous	The function returns when the provider lifts flow-control restrictions. The kOTFlowErr result is never returned.	The function returns immediately. The kOTFlowErr result might be returned.
Asynchronous	The function returns immediately. The kOTFlowErr result might be returned.	The function returns immediately. 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 data is sent. See "Option Management"(page 165).

SEE ALSO

Table 4-4 (page 95).

OTRcv

Reads data sent using a connection-oriented transactionless endpoint.

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 buffer where the incoming data is to be copied. You must allocate this buffer before you call the function.

`nbytes` A 32-bit value specifying the size of the buffer in bytes.

`flags` A 32-bit bitmapped value specifying, on return, whether the data being sent is expedited (`T_EXPEDITED`) and whether more data remains to be received (`T_MORE`).

function result A positive integer specifying the number of bytes received or a negative integer specifying an error code. See Appendix B(page 785), and Discussion.

DISCUSSION

You call the `OTRcv` function to read data sent by the peer to which you are connected. If the `OTRcv` function succeeds, it returns a positive integer (`OTResult`) 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 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. If the endpoint supports TSDUs and the T_MORE bit is set in the flags parameter when the function returns, this means that the buffer you allocated is too small to contain the entire TSDU 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. Each call to this function that returns with the T_MORE flag set means that you must call the function again to get more of the TSDU. When you have read the entire TSDU, the OTRcv function returns with the T_MORE flag not set.

If the OTRcv function returns and the T_EXPEDITED bit is set in the flags parameter, this means that you have read expedited data. If the number of bytes in the ETSDU 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 ETSDU.

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.

If the endpoint is in synchronous blocking mode, the endpoint waits for data if none is currently available.If the endpoint is in asynchronous mode or is not blocking, the function returns with the kOTNoDataErr result if no data is available. For more information on notifier functions and event codes, see MyNofierCallback function(page 413) and “Event Codes”(page 383). 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 (page 449). 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.

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 were in the process of 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 events are cleared, Open Transport cannot send you another T_DATA event (or any other event other than a T_DISCONNECT event).

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. See "Option Management" (page 165) for information on setting this 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 ASP is described at greater length in the section "Using Connection-Oriented Transaction-Based Service" (page 125).

SEE ALSO

Table 4-4 (page 95).

OTSndRequest

Sends a request to a connection-oriented transaction-based endpoint.

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 434) that contains information about the request, options for this request, and the transaction ID of the request. You must allocate buffers for this information and specify their size.

The `req->data.buf` field is apointer to a buffer that contains the request. Set the `req->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.

The `req->opt.buf` field is a pointer to a buffer that contains option values you want to negotiate for this request. Set the `req->opt.len` field to the size of the option data. Set the `opt.len` field to 0 if there are no options.

The `req->sequence` field specifies a unique non-zero number of your choice that identifies this request.

`reqFlags` A bitmapped 32-bit value 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 bitwise `OR` operator to set both bits.

function result An error code. See Appendix B(page 785), and Discussion.

DISCUSSION

You use the OTSndRequest function to initiate a transaction for a connection-oriented transaction-based endpoint. Call the OTRcvReply function (page 507) to read the reply to your request.

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 synchronous 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 asynchronous or 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 (page 449). 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 kOTFlowErr result is never returned.	The function returns if flow-control restrictions are in effect or the request data has been sent to the protocol. The kOTFlowErr result might be returned.
Asynchronous	The function returns immediately. The kOTFlowErr result might be returned.	The function returns immediately. The kOTFlowErr result might be returned.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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);
```

PARAMETERS

`ref`	The endpoint reference of the endpoint reading the request.
`req`	A pointer to a `TRequest` structure (page 434) that contains information, on return, about the incoming request. You must allocate buffers in which this information can be stored and specify the maximum size of each buffer. The `req->data.buf` field is a pointer to a buffer in which the incoming request is stored. Set the `req->data.maxlen` field to the maximum size of the request. The `req->opt.buf` field is a pointer to a buffer in which the options specified by the requester are stored. Set the `req->opt.maxlen` field to the maximum size of option data. The value for the `req->sequence` field is generated by the endpoint provider. You need to save this value and use it for the `sequence` field when sending a reply.
`reqFlags`	A bitmapped 32-bit value specifying, on return, whether there is more request data coming (`T_MORE`) and whether the provider is going to acknowledge replies (`T_ACKNOWLEDGED`).
function result	An error code. See Appendix B(page 785), and Discussion.

DISCUSSION

You use the `OTRcvRequest` function to read an incoming request. After reading the request, you can use the `OTSndReply` function (page 504) to reply to that request or the `OTCancelRequest` function (page 510) 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 the endpoint is in synchronous mode and is blocking, the `OTRcvRequest` function returns only when a request arrives. The `OTRcvRequest` function returns either the next unread request or the `kOTNoDataErr` result if there are no pending requests.

When a request arrives, the provider generates a `T_REQUEST` event. You can poll for this event using the `OTLook` function (page 449) 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. For more information on notifier functions and event codes, see MyNotifierCallback function(page 413) and "Event Codes"(page 383).

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.buf field contains no information for these additional calls.

The flags parameter can also have the T_PARTIALDATA bit set. In this case, the request data being received is only partial; there is more data coming, but it has not arrived. The difference between T_MORE and T_PARTIALDATA is that the first indicates that there is more data and the next call to OTRcvRequest will read that data, while the second does not gurantee that the next read will get the data. The T_PARTIALDATA bit is only set on the first call to the function. All subsequent calls to OTRcvRequest are guaranteed to continue reading the partial data in question until the function returns with the T_MORE flag clear.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

OTSndReply

Replies 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 replyFlags );
```

PARAMETERS

`ref` The endpoint reference of the endpoint reading the request.

`reply` A pointer to a `TReply` structure (page 434) that specifies the reply data being sent, the transaction ID for this transaction, and any options you want to set.

The `reply->data.buf` field is a pointer to the buffer containing the reply. Set the `reply->data.len` field to the size of the reply. The size of the reply must not exceed the value specified for the `tsdu` field of the `TEndpointInfo` structure for this endpoint.

The `reply->opt.buf` field is a pointer to the buffer that contains the options you want to set. Set the `reply->opt.len` field to the size of the option data or to 0 if you don't want to specify any options.

The `reply->sequence` field specifies the transaction ID of the request to which you are replying. (This value was returned in the `req->sequence` field by the `OTRcvRequest` function.)

`replyFlags` A bitmapped 32-bit value 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).

function result An error code. See Appendix B(page 785), and Discussion.

DISCUSSION

You use the `OTSndReply` function to reply to a request you have read using the `OTRcvRequest` function (page 502). 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 , 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 synchronous blocking mode, the OTSndReply function returns after it has sent the reply. If the endpoint is in synchronous 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 (page 449) 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. For more information on notifier functions and event codes, see MyNotifierCallback function(page 413) and “Event Codes”(page 383).

If the reply was not sent successfully (that is, timed out—for acknowledged requests), the kETIMEDOUTErr result code is returned. For unacknowledged requests, the T_REPLYCOMPLETE event is still generated for asynchronous clients so that the logic is the same for both acknowledged and unacknowledged requests.

The cookie parameter passed to the notifier is set to the reply parameter of the original request. In cases where the T_MORE flag is used to send the reply in muliple chunks, the first TReply* is used.

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.

	Blocking	Nonblocking
Asynchronous	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 `kOTFlowErr` result might be returned.	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 `kOTFlowErr` result might be returned.

SEE ALSO

The `OTRcvReply` function (page 507).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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* replyFlags);
```

PARAMETERS

`ref` The endpoint reference of the endpoint reading the reply.

`reply` A pointer to a `TReply` structure (page 434) that specifies where to store the reply information. You must allocate buffers to contain the reply data, option values, and the transaction ID of the request to which the reply is being sent, and you must specify the maximum size of this data.

The `reply->data.buf` field points to a buffer in which the reply data is to be stored. Set the `reply->data.maxlen` field to the maximum size of the reply data.

The `reply->opt.buf` field points to a buffer in which option values are stored. Set the `reply->opt.maxlen` field to the maximum size of this data.

The `reply->sequence` field specifies the transaction ID of the transaction to which the reply is sent. If you have sent out multiple requests, you can examine this field to match replies to requests.

`replyFlags` A bitmapped 32-bit value 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.

function result An error code. See Appendix B(page 785), and Discussion.

DISCUSSION

You use the `OTRcvReply` function to read the reply to a request that you sent using the `OTSndRequest` function (page 499).

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 (page 449) 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 available data has been read. The T_PARTIALDATA event might be returned if the entire reply is not available.

If the T_MORE bit is set in the replyFlags 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 replyFlags parameter, this means that the data you are about to read with the OTRcvReply function does not constitute the entire reply and that more data is coming, but it has not yet arrived. You must call the function again to read more, 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.

SEE ALSO

The OTSndRequest function (page 499).

Table 4-4 (page 95).

OTCancelRequest

Cancels an outstanding request as defined by a call to the OTSndRequest function.

C INTERFACE

```
OSStatus OTCancelRequest(EndpointRef ref,
                         OTSequence sequence);
```

C++ INTERFACE

```
OSStatus TEndpoint::CancelRequest(OTSequence sequence);
```

PARAMETERS

ref The endpoint reference of the endpoint that has sent the request being cancelled.

sequence A 32-bit value, 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 for the endpoint. If you specify an invalid sequence number, the provider does nothing .

function result An error code. See Appendix B(page 785).

DISCUSSION

When you make a call to the OTSndRequest function (page 499), 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 endpoint is acknowledging sends, the provider generates a `T_MEMORYRELEASED` event for each freed buffer.

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` (page 511) to cancel an *incoming* request.

SEE ALSO

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

OTCancelReply

Cancels an outstanding call to the `OTRcvRequest` function.

C INTERFACE

```
OSStatus OTCancelReply(EndpointRef ref,
                       OTSequence sequence);
```

C++ INTERFACE

```
OSStatus TEndpoint::CancelReply(OTSequence sequence);
```

PARAMETERS

`ref` The endpoint reference of the endpoint that has sent the request being canceled.

`sequence` A 32-bit value, 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 for the endpoint. If you specify an invalid sequence number, the provider does nothing.

function result An error code. See Appendix B(page 785).

DISCUSSION

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 endpoint is acknowledging sends, the provider generates a `T_MEMORYRELEASED` event for each freed buffer.

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 (page 510) to cancel an *outgoing* request.

SEE ALSO

The `OTSndRequest` function (page 499).

"AppleTalk Reference" (page 721).

Table 4-4 (page 95).

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 433) 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.

The `call->addr` field and the `call->opt` field are reserved and should be set to 0.

The `call->udata.buf` field is a pointer to a buffer containing data that you want to send with the disconnection request. Set the `call->udata.len` field to the size of the data. The amount of data should not exceed the limits supported by the endpoint, as returned in the `discon` field of the `TEndpointInfo` structure (page 426).

The `call->sequence` field should be set to a valid value to identify the request if you are using this function to reject a connection request. This field is ignored if you are using this function to tear down a connection.

function result An error code. See Appendix B(page 785).

DISCUSSION

You can use the `OTSndDisconnect` function to tear down a connection or to reject incoming connection requests. Whenever possible, use the function `OTSndOrderlyDisconnect` to tear down a connection.

If the endpoint is in synchronous mode, the function returns when the operation is complete. 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 sent a `T_DISCONNECTCOMPLETE` event when the process completes. The `cookie` parameter contains the `call` parameter.

If you have not installed a notifier function, you cannot determine when this function completes. For more information on notifier functions and event codes, see `MyNotifierCallback` function(page 413) and "Event Codes"(page 383).

SEE ALSO

"TCP/IP Services" (page 237).

"AppleTalk Reference" (page 721).

"Terminating a Connection" (page 112).

Table 4-4 (page 95).

OTRcvDisconnect

Identifies the cause of a connection break or of a connection rejection 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 (page 435) that is filled ou;with disconnection data, a reason for the disconnection, and a connection request sequence number.
function result	An error code. See Discussion.

DISCUSSION

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 `NULL` 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 did not supply a buffer that was large enough to hold the disconnection data.

SEE ALSO

The `OTSndDisconnect` function (page 513).

"TCP/IP Services" (page 237).

"AppleTalk Reference" (page 721).

"Terminating a Connection" (page 112).

Table 4-4 (page 95).

OTSndOrderlyDisconnect

Initiates 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.
function result	An error code. See Appendix B(page 785).

DISCUSSION

You call the `OTSndOrderlyDisconnect` function to initiate an orderly disconnect 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 until the peer calls 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 (page 426) has the value `T_COTS_ORD` or `T_TRANS_ORD`.

The `OTSndOrderlyDisconnect` function behaves exactly the same in all modes of operation.

SEE ALSO

The `OTSndDisconnect` function (page 513).

"Terminating a Connection" (page 112).

Table 4-4 (page 95).

OTRcvOrderlyDisconnect

Clears an incoming orderly disconnect event.

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.
function result	An error code. See Appendix B(page 785).

DISCUSSION

You call the OTRcvOrderlyDisconnect function to acknowledge the receipt of an orderly disconnect event. After using the OTRcvOrderlyDisconnect function, there will not be any more data to receive. Attempts to receive data will 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 (page 516).

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. If there is a disconnection request pending, the function returns the kOTNoError result.

The OTRcvOrderlyDisconnect function is only supported by those protocols that have a servtype field of the TEndpointInfo structure with a value T_COTS_ORD or T_TRANS_ORD.

SEE ALSO

“Terminating a Connection” (page 112).

Table 4-4 (page 95).

CHAPTER 23

Programming With Open Transport Reference

Contents

This chapter describes the Open Transport functions that enable you to schedule system tasks, timer, and deferred tasks, and to enter and leave interrupt level processing. This chapter also describes functions you can use to idle or delay your computer; these functions are included for compatibility with UNIX-based programs.

Data Types

Callback Function

You can use the `OTProcessProcPtr` prototype to declare a callback routine. You can then pass this routine as a parameter to the Open Transport functions used to create a system, timer, or a deferred task. 4The routine defines the routine that you want Open Transport to execute at that time. The routine takes one argument, which contains user-defined context information.

```
typedef pascal void (*OTProcessProcPtr)(void* arg);
```

For information about creating your own callback routine, see the description of the function `MyOTProcessProc` (page 541). An `OTProcessProcPtr` is never a Universal Proc Pointer.

Functions

This section describes the functions you use when managing the special processing needs of your application—that is, whether code runs at system task time, at deferred task time, at interrupt time, or at some predefined interval.

Checking Synchronous Calls

Open Transport provides the `OTCanMakeSyncCall` function so that you can check whether you can call a synchronous function.

OTCanMakeSyncCall

Checks whether you can call a synchronous function.

C INTERFACE

```
Boolean OTCanMakeSyncCall();
```

C++ INTERFACE

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

PARAMETERS

function result The function returns `true` if you can call a synchronous function.

DISCUSSION

The `OTCanMakeSyncCall` function returns a value of `true` if you can make a synchronous call to Open Transport. The function returns a value of `false` if you cannot make a synchronous call.

SEE ALSO

See “Providers” (page 61)for additional information about synchronous execution.

OTIsAtInterruptLevel

Checks whether Open transport is currently executing at interrupt level.

C INTERFACE

```
Boolean OTIsAtInterruptLevel(void);
```

C++ INTERFACE

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

PARAMETERS

function result The function returns `true` if Open Transport is at interrupt level.

OTCanLoadLibraries

Checks whether Open Transport can load libraries.

C INTERFACE

```
Boolean OTCanLoadLibraries(void);
```

C++ INTERFACES

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

PARAMETERS

function result The function returns `true` if you can load libraries.

Working With System Tasks

Open Transport provides several functions that allow you to defer execution of your code until system task time.

OTCreateSystemTask

Creates a reference to a task that can later be scheduled to run at system task time.

C INTERFACE

```
long OTCreateSystemTask(OTProcessProcPtr proc,
                        void* arg)
```

C++ INTERFACE

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 context pointer for your use. Open Transport simply passes the value as the original to the procedure specified by the `proc` parameter. Pass `NULL` if you do not want any 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.
function result	A reference that should be used when scheduling or destroying the task. A value of 0 indicates that there is not enough memory to allocate the necessary data.

DISCUSSION

The `OTCreateSystemTask` function creates a system task that you can schedule for execution at system task time using the `OTScheduleSystemTask` function. The task gets a pointer to the callback function specified by the `proc` parameter, and its argument specified by the `arg` parameter. (For 68000 code, when Open Transport executes the callback function, it restores the A5 global world to what it was when you originally called the `OTCreateSystemTask` function.)

It is important that you call the function `OTDestroySystemTask` when you are done with processing your system tasks. Open Transport does not automatically deallocate the memory and resources used to schedule system tasks when you close Open Transportjuok.

IMPORTANT

Never pass a universal procedure pointer as the `proc` parameter for this call. ▲

SEE ALSO

The `OTScheduleSystemTask` function (page 525).

The `OTDestroySystemTask` function (page 528).

The `OTCancelSystemTask` function (page 527).

OTScheduleSystemTask

Schedules a task for execution at system task time.

C INTERFACE

```
Boolean OTScheduleSystemTask(long stCookie)
```

C++ INTERFACE

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

PARAMETERS

`stCookie` A reference that identifies the task to be scheduled.

function result A result of `true` means that the system task is scheduled to run. If the function returns `false` and the `stCookie` parameter has a valid value (i.e. 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.

DISCUSSION

The `OTScheduleSystemTask` function schedules for execution at system task time the task associated with the `stCookie` parameter, which is a reference to a system task created 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 make multiple calls to the `OTScheduleSystemTask` function before the task is executed, additional tasks are not scheduled; only one instance of each unique task is scheduled at a time.

▲ **WARNING**

Open Transport does not keep track of outstanding system task requests. It is the caller's responsibility to ensure that all outstanding system task requests have either executed or have been cancelled. If this is not done, and the calling process terminates before the system task executes, the processor will crash when it attempts to execute the task. ▲

SPECIAL CONSIDERATIONS

You can call this Open Transport function at hardware interrupt time, but you must precede it by calling the `OTEnterInterrupt` function and you must follow it by calling the `OTLeaveInterrupt` function.

SEE ALSO

The `OTCreateSystemTask` function (page 524).

The `OTDestroySystemTask` function (page 528).

The `OTCancelSystemTask` function (page 527).

The `OTEnterInterrupt` function (page 537).

The `OTLeaveInterrupt` function (page 540).

OTCancelSystemTask

Cancels a system task that you have scheduled.

C INTERFACE

```
Boolean OTCancelSystemTask(long stCookie)
```

C++ INTERFACE

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

PARAMETERS

`stCookie`	A reference value that identifies the task to be canceled.
function result	A result of `true` means that the scheduling was canceled. If the function returns `false`, then either the function was not scheduled, or it was too late to cancel it. If the `stCookie` parameter value is invalid (a value of 0), the function returns `false` and does nothing.

DISCUSSION

The `OTCancelSystemTask` function cancels a task that was scheduled with the `OTScheduleSystemTask` function to run at system task time.

SPECIAL CONSIDERATIONS

You can call this Open Transport function a thardware interrupt time, but you must precede it by calling the `OTEnterInterrupt` function, and you must follow it by calling the `OTLeaveInterrupt` function.

SEE ALSO

The `OTCreateSystemTask` function (page 524).

The `OTScheduleSystemTask` function (page 525).

The `OTDestroySystemTask` function (page 528).

The `OTEnterInterrupt` function (page 537).

The `OTLeaveInterrupt` function (page 540).

OTDestroySystemTask

Destroys a system task created with the `OTCreateSystemTask` function.

C INTERFACE

```
OSStatus OTDestroySystemTask(long stCookie)
```

C++ INTERFACE

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

PARAMETERS

`stCookie`	A reference value that identifies the system task to be destroyed.
function result	The result `kOTNoError`.

DISCUSSION

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 executed 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

The `OTCreateSystemTask` function (page 524).

The `OTScheduleSystemTask` function (page 525).

The `OTCancelSystemTask` function (page 527).

Working With Timer Tasks

Use the following function to create, scedule, cancel, and destroy a timer task. Open Transport executes these tasks at deferred task time.

IMPORTANT

You cannot call these functions from 68000 code running on a Power PC. ▲

OTCreateTimerTask

Creates a task to be scheduled.

C INTERFACE

```
pascal long OTCreateTimerTask(OTProcessProcPtr proc, void* arg);
```

C++ INTERFACES

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

PARAMETERS

`proc` A procedure pointer referencing the task to be created.

arg — A pointer to a user-defined value to be passed to the task when it executes.

function result — A 32-bit value identifying the task that was created. You must pass this value for the `timerTask` parameter of the `OTScheduleTimerTask` function.

DISCUSSION

The `OTCreateTimerTask` function creates a task that you can schedule later with the function `OTScheduleTimerTask`.

SEE ALSO

The `OTScheduleTimerTask` function (page 530).

The `OTCancelTimerTask` function (page 531).

The `OTDestroyTimerTask` function(page 532).

OTScheduleTimerTask

Schedules a timer task to be executed at the specified time.

C INTERFACE

```
pascal Boolean OTScheduleTimerTask(long timerTask,
                          OTTimeout milliSeconds);
```

C++ INTERFACES

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

PARAMETERS

`timerTask` — A 32-bit value identifying the task you want to schedule. This is the value returned by the `OTCreateTimerTask` when you created the task.

`OTTimeout` The time, in milliseconds, that should elapse between when you call `OTScheduleTimerTask` and when the function executes.

function result The result is `true` if the timer task is scheduled.

DISCUSSION

The `OTScheduleTimerTask` schedules the execution of a task created with the `OTCreateTimerTask` function.

▲ **WARNING**
Calling this function periodially to schedule a task will cause drift. ▲

SEE ALSO

The `OTCancelTimerTask` function (page 531).

The `OTDestroyTimerTask` function(page 532).

OTCancelTimerTask

Cancels a task that was already scheduled for execution.

C INTERFACE

```
pascal Boolean OTCancelTimerTask(long timerTask);
```

C++ INTERFACE

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

PARAMETERS

`timerTask` A 32-bit value identifying the task you want to cancel. This is the value you passed to `OTScheduleTimerTask` when you scheduled the task.

function result The result is `true` if the timer task was cancelled.

DISCUSSION

The `OTCancelTimerTask` cancels the execution of a task scheduled with the `OTScheduleTimerTask` function.

SEE ALSO

The `OTScheduleTimerTask` function (page 530).

The `OTDestroyTimerTask` function(page 532).

OTDestroyTimerTask

Disposes of a timer task.

C INTERFACE

```
pascal void OTDestroyTimerTask(long TimerTask);
```

C++ INTERFACE

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

PARAMETERS

`timerTask` A 32-bit value identifying the timer task to be destroyed. This value is returned by the `OTCreateTimerTask` function.

function result The result `kOTNoError`.

DISCUSSION

The `OTDestroyTimerTask` function deallocates memory and resources used to store and schedule a timer task.

SEE ALSO

The `OTScheduleTimerTask` function (page 530).

The `OTCancelTimerTask` function (page 531).

Working With Deferred Tasks

Open Transport provides several functions that allow you to handle deferred tasks in your code. You should use these functions rather than the standard Deferred Task Manager function `DTInstall` to create a deferred task that includes calls to Open Transport functions.

OTCreateDeferredTask

Creates a reference to a task that can be scheduled to run at deferred task time.

C INTERFACE

```
long OTCreateDeferredTask(OTProcessProcPtr proc,
                          void* arg);
```

C++ INTERFACE

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 context pointer for your use. 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.

function result A reference that you can use to identify a task when calling the `OTScheduleDeferredTask` function or the `OTDestroyDeferredTask` function. If the return value is 0, then there is not enough memory to allocate the necessary data.

DISCUSSION

The `OTCreateDeferredTask` function creates a deferred task that you can schedule for execution at deferred task time using the `OTScheduleDeferredTask` function.

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 allows you to call Open Transport functions.

SEE ALSO

The `OTScheduleDeferredTask` function (page 534).

The `OTDestroyDeferredTask` function (page 536).

OTScheduleDeferredTask

Schedules a task for execution at deferred task time.

C INTERFACE

```
Boolean OTScheduleDeferredTask(long dtCookie)
```

C++ INTERFACE

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

PARAMETERS

`dtCookie` A reference that identifies the task to be scheduled. This reference is returned by the `OTCreateDeferredTask` function used to create the task.

function result The result is `true` if the deferred task is scheduled successfully. If the result is `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.

DISCUSSION

The `OTScheduleDeferredTask` function schedules the task (specified by the `dtCookie` parameter) for execution at the next deferred task time.

You can call the `OTScheduleDeferredTask` 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 make multiple calls to the `OTScheduleDeferredTask` function before the task is executed, additional tasks are not scheduled; only one instance of each unique task is scheduled at a time.

▲ **WARNING**
Open Transport does not keep track of outstanding deferred task requests. It is the caller’s responsibility to ensure that all outstanding deferred task requests have either executed or have been cancelled. If this is not done, and the calling process terminates before the deferred task executes, the processor will crash when it attempts to execute the task. ▲

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

The `OTScheduleInterruptTask` function (page 538).

The `OTCreateDeferredTask` function (page 533).

The `OTDestroyDeferredTask` function (page 536).

The `OTEnterInterrupt` function (page 537).

The `OTLeaveInterrupt` function (page 540).

OTDestroyDeferredTask

Destroys a deferred task created with the `OTCreateDeferredTask` function.

C INTERFACE

```
OSStatus OTDestroyDeferredTask(long dtCookie)
```

C++ INTERFACE

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

PARAMETERS

`dtCookie` A reference that identifies the task to be destroyed.

function result The function returns the result `kEAGAINErr` when you try to destroy the task from within an interrupt service routine or within another deferred task. If `dtCookie` is invalid (a value of 0), the function returns `kOTNoError` and does nothing.

DISCUSSION

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.

SEE ALSO

The `OTCreateDeferredTask` function (page 533).

The `OTScheduleDeferredTask` function (page 534).

Entering and Leaving Hardware Interrupt Time

Open Transport provides functions that you use to inform Open Transport that you are entering and leaving hardware interrupt time.

OTEnterInterrupt

Notifies Open Transport that you are about to call an Open Transport function from a hardware interrupt.

C INTERFACE

```
void OTEnterInterrupt(void)
```

C++ INTERFACE

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

DISCUSSION

The `OTEnterInterrupt` function informs Open Transport it is at hardware interrupt time. This allows Open Transport to schedule network activity more intelligently. You must use this function before calling many of the Open Transport functions from hardware interrupt time, for example, from a VBL or Time Manager task, or from a File Manager or Device Manager I/O completion routine. There are only a limited number of Open Transport functions that you can actually call at hardware interrupt time; these are listed in Appendix C.

You must call the `OTLeaveInterrupt` function before you leave interrupt time.

Note
The OTEnterInterrupt function was provided to enable callers of the OTScheduleDeferredTask function a means to let Open Transport know that the scheduling was occurring at interrupt time. With the addition of the OTScheduleInterruptTask function (which calls OTEnterInterrupt internally), it is no longer necessary to call the OTEnterInterrupt function. There are cases, however, when it might be advisable to call the OTEnterInterrupt and OTScheduleDeferredTask functions in sequence; for example, when you want to schedule two deferred tasks consecutively, or when you are calling a common function which uses OTScheduleDeferredTask and which is sometimes called at interrupt time and sometimes not. ◆

SPECIAL CONSIDERATIONS

If you are running 680x0-based code, you must be sure that your A5 world is set correctly before making this call (that is, that it has the same value it had when you called the InitOpenTransport or the InitOpenTransportUtilities function).

SEE ALSO

The OTLeaveInterrupt function (page 540).

“Hardware Interrupt Level” (page 136) .

OTScheduleInterruptTask

Schedules a task for execution at hardware interrupt time.

C INTERFACE

```
Boolean OTScheduleInterruptTask(long dtCookie)
```

C++ INTERFACE

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

PARAMETERS

`dtCookie` A reference that identifies the task to be scheduled. This reference is returned by the `OTCreateDeferredTask` function, which you use to create the task.

function result The result is `true` if the deferred task is scheduled. 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.

DISCUSSION

The `OTScheduleInterruptTask` function schedules the task associated with the `dtCookie` parameter for execution at deferred task time.This function calls the `OTEnterInterrupt` and the `OTLeaveInterrupt` functions, so you do not need to make separate calls to those functions when calling the `OTScheduleInterruptTask` function.

The `OTScheduleInterruptTask` function behaves exactly like the function `OTScheduleDeferredTask`. You should call it instead of calling the `OTScheduleDeferredTask` function if you are calling from hardware interrupt time.

When Open Transport calls your callback function, it passes it the `arg` parameter you specified when you created the task. (For 68000 code, Open Transport also restores the A5 global world to what it was when you originally called `OTCreateDeferredTask`.)

You can call this function 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 will be scheduled at a time.

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 functions. This allows Open Transport to adapt to changes in the underlying operating system without affecting your code.

▲ **WARNING**
Open Transport does not keep track of outstanding deferred task requests. It is the caller's responsibility to ensure that all outstanding deferred task requests have either executed or have been cancelled. If this is not done, and the calling process terminates before the deferred task executes, the processor will crash when it attempts to execute the task. ▲

SPECIAL CONSIDERATIONS

You can call this Open Transport function at hardware interrupt time without calling the `OTEnterInterrupt` function first and the `OTLeaveInterrupt` function afterward.

SEE ALSO

The `OTCreateDeferredTask` function (page 533).

The `OTDestroyDeferredTask` function (page 536).

OTLeaveInterrupt

Notifies Open Transport that you are done calling Open Transport functions from a hardware interrupt.

C INTERFACE

```
void OTLeaveInterrupt(void)
```

C++ INTERFACE

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

DISCUSSION

The OTLeaveInterrupt function informs Open Transport that you are no longer calling Open Transport from within a routine that runs at hardware interrupt time. You call this function after you call the last Open Transport function you want to call from within the interrupt, but before you return from your interrupt handler.

Every call to the OTEnterInterrupt function must be matched by a call to the OTLeaveInterrupt function.

SPECIAL CONSIDERATIONS

Do not make this call without having already called the OTEnterInterrupt function.

For 680x0 based code, 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

The OTEnterInterrupt function (page 537).

"Hardware Interrupt Level" (page 136).

Application-Defined Functions

This section describes the user-defined callback function that you can schedule to run as a system task, timer task, or deferred task.

MyOTProcessProc

Defines a callback function.

```
pascal void MyOTProcessProc (void* arg);
```

arg	A pointer to user-defined context information.

DISCUSSION

The user-defined function MyOTProcessProc defines a function that Open Transport executes as a system task, deferred task, or timer task.

- To have the function execute at system task time, you must first call the function OTCreateSystemTask (page 524), passing the address of the function for the proc parameter and a pointer to any context information for the arg parameter. The function OTCreateSystemTask returns a reference to the newly created task which you then pass as the stCookie parameter to the OTScheduleSystemTask function (page 525), which schedules the task to run. When Open Transport executes the task, it passes it the value you specified for the arg parameter when you created the task.
- To have the function execute at some specified interval (timer task), you must first call the function OTCreateTimertask(page 529), passing the address of the function for the proc paramter and a pointer to any context information for the arg parameter. The OTCreateTimertask function returns a reference to the newly created task which you can then pass as the timerTask parameter to the OTScheduleTimerTask function (page 530), which you use to schedule the task. Use the OTCancelTimerTask function (page 531) to cancel the task and the OTDestroyTimerTask function (page 532) to dispose of it.
- To have the function execute at deferred task time, you must first call the function OTCreateDeferredTask (page 533), passing the address of the function for the proc parameter and a pointer to any context information for the arg parameter. The function OTCreateDeferredTask returns a reference to the newly created task which you then pass as the dtCookie parameter to the OTScheduleDeferredTask function (page 534), which schedules the task to run. When Open Transport executes the task, it passes it the value you specified for the arg parameter when you created the task.

SEE ALSO

See also “Programming With Open Transport” (page 129) for additional information about system tasks, timer tasks, and deferred tasks.

Mappers Reference

Contents

This chapter describes the constants, 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 chapter "Providers Reference" (page 383).

Constants and Data Types

This section describes the constants and data types used by mapper functions.

Error-Checking Constant

Open Transport provides a constant that you can use to initialize the private mapper reference structure. You can then check that the current mapper reference is valid before passing it to mapper functions. The constant is defined as follows:

```
#define     kOTInvalidMapperRef ((MapperRef)0)
```

The TRegisterRequest Structure

You use the `TRegisterRequest` structure to specify the entity name you want to register using the `OTRegisterName` function (page 554) and, optionally, to specify its address.

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

```
struct TRegisterRequest
{
    TNetbuf name;
    TNetbuf addr;
    OTFlags flags;
};
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 buffer.

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 buffer. 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.

flags A field used to control registration. Normally, this field is set to 0 for default registration behavior. See the documentation for the naming service you are using for more information.

The TRegisterReply Structure

You use the TRegisterReply structure to store information returned by the OTRegisterName function (page 554).

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 specifies 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 size of the buffer.

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 entity name to be looked up by the OTLookupName function (page 559) 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;
    TNetbuf     addr;
    UInt32      maxcnt;
    OTTimeout   timeout;
    OTFlags     flags;
};
    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 to be stored. You should normally supply 0 for addr.len. This causes the provider to use internal defaults for the starting point of the search. For a protocol family such as 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 or do not expect to exceed a specific number, you should specify this number to obtain faster execution. Otherwise, set this field to 0xffff ffff; in this case, the timeout value will control the lookup.

timeout
A long specifying the amount of time, in milliseconds, that should elapse before Open Transport gives up searching for a name. Specify 0 to leave the timeout value up to the underlying naming system.

The TLookupReply Structure

You use the TLookupReply structure to store information passed back to you by the OTLookupName function (page 559). The information includes both a pointer to a buffer (containing 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 the buffer into 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 size of the buffer.

rspcount
: A long specifying, on return, the number of names found.

The TLookupBuffer Structure

The TLookupBuffer structure defines the format of entries in the buffer passed back in the reply parameter of the OTLookupName function (page 559). When you parse the buffer in which the OTLookupName function places the names it has found, you can cast it as a TLookupBuffer structure. Figure 6-1 (page 154) 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];
};
typedef struct TLookupBuffer TLookupBuffer;
```

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
: the first byte of the address to which the entity whose name follows (in the buffer) is bound.

Functions

This section describes functions that you use only with mappers to manage the mapping of names to protocol 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. However, 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 Reference” (page 383).

You can also use general provider functions with mappers. You use these functions to change a function’s modes of operation (for example, to blocking). General provider functions are described in the reference section of the chapter “Providers”(page 383).

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 the mapper provider.

OTOpenMapper

Creates a synchronous mapper provider and returns a mapper reference.

C INTERFACE

```
MapperRef OTOpenMapper(OTConfiguration* config,
                       OTOpenFlags oflag,
                       OSErr* err);
```

C++ INTERFACE

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 disposes of the configuration structure before returning.

`oflag` Reserved; must be set to 0.

`err` A pointer to the result code for this function.

function result A reference that you can use when calling other mapper functions.

DISCUSSION

The `OTOpenMapper` function opens a mapper having the configuration specified by the `config` parameter. For additional informationon Mappers and configurations see "Getting Started With Open Transport" (page 31) and the documentation provided for the mapper 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 `kOTInvalidMapperRef`.

A mapper created by the `OTOpenMapper` function operates in synchronous mode, and does not block. To change the mapper's default modes of operation, you can call the `OTSetAsynchronous` and the `OTSetBlocking` function.

You can open multiple mappers using identical or different configurations. The different mappers can be distinguished by the mapper reference.

To delete a mapper, call the OTCloseProvider function (page 392). For additional information about a mapper's mode of operations, see "Setting Modes of Operation for Mappers" (page 151) .

SPECIAL CONSIDERATIONS

Because the OTOpenMapper function executes synchronously, you should only call this function at system task time.

The OTOpenMapper function destroys the configuration structure passed in the config parameter. If you want to use the same configuration to open additional mappers, you must copy the configuration structure by calling the OTCloneConfiguration function(page 378).

SEE ALSO

The OTAsyncOpenMapper function (page 550).

The OTSetAsynchronous function(page 395) .

The OTSetBlocking function (page 398).

OTAsyncOpenMapper

Creates an asynchronous mapper and installs a notifier function for the mapper provider.

C INTERFACE

```
OSStatus OTAsyncOpenMapper (OTConfiguration* config,
                            OTOpenFlags oflag,
                            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 mapper's characteristics. You obtain a value for the `config` parameter by calling the `OTCreateConfiguration` function. The `OTAsyncOpenMapper` function disposes of the configuration before returning.
`oflag`	Reserved; must be set to 0.
`proc`	A pointer to a notifier function for this mapper.
`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` parametert o pass to your notifier function information about your mapper.
function result	An error code. See Appendix B(page 785) for more information.

DISCUSSION

The `OTAsyncOpenMapper` function opens a mapper having the configuration specified by the `config` parameter. For additional information on mappers and configurations see “Getting Started With Open Transport” (page 31) and the documentation provided for the mapper protocol you are using. The `OTAsyncOpenMapper` function runs asynchronously, returning 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. For more information on notifier functions and event codes see “Using Notifier Functions”(page 405) and “Event Codes”(page 383).

A mapper created by the `OTAsyncOpenMapper` function operates in asynchronous mode and does not block . For more information about a mapper's mode of operations, see the section “Setting Modes of Operation for Mappers” (page 151) .

You can open multiple mappers using identical or different configurations. 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.

To close a mapper use the `OTCloseProvider` function(page 392).

SPECIAL CONSIDERATIONS

The `OTAsyncOpenMapper` function destroys the configuration value passed in the config parameter. You cannot use the same configuration to open multiple mappers. To copy the configuration for use when opening another mapper, you must call the `OTCloneConfiguration` function(page 378).

SEE ALSO

The `OTOpenMapper` function (page 550).

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 a name on the network.

C INTERFACE

```
OSStatus OTRegisterName (MapperRef ref,
                       TRegisterRequest* req,
                         TRegisterReply* reply);
```

C++ INTERFACE

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

PARAMETERS

`ref` A mapper reference.

`req`	A pointer to a `TRegisterRequest` structure (page 545) that specifies the entity name you want to register and the protocol address it maps to.
`reply`	A pointer to a `TRegisterReply` structure (page 546) that returns the actual protocol address registered and an ID that can be used to delete the name.
function result	An error code. See Appendix B(page 785) and the discussion below for more information.

DISCUSSION

If the mapper protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` functions supports dynamic name and address registration, you can use the `OTRegisterName` function to make a name visible on the network to other network devices.

Note
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(page 441). 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.

If the mapper is in synchronous mode, the function does not return until the operation is complete. If the mapper is in asynchronous mode, the `OTRegisterName` function returns immediately. When the operation completes execution, the mapper provider issues the `T_REGNAMECOMPLETE` code event. The `cookie` parameter to the notification routine has the value of the `reply` parameter passed to the `OTRegisterName` function. For more information on notifier functions and event codes see “Using Notifier Functions”(page 405) and “Event Codes”(page 383).

If the name was already registered, the function returns the result kOTAddressBusyErr. If the reply->addr.maxlen field is set to 0, then the address is not filled in and the result kOTNoError result is returned. If this field is not set to 0, the result kOTBufferOverflowErr is returned if the allocated buffer is not large enough to hold the address.

SEE ALSO

The OTLookupName function (page 559).

The OTDeleteName function (page 556).

The OTDeleteNameByID function (page 557) .

The OTOpenMapper(page 550).

The AsyncOpenMapper(page 552).

OTDeleteName

Removes a previously registered entity name.

C INTERFACE

```
OSStatus OTDeleteName (MapperRef ref,
                       TNetbuf* name);
```

C++ INTERFACE

```
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.

function result If the name is not found, the function returns the result `kOTNoAddressErr`.

DISCUSSION

If the mapper protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` functions supports dynamic name and address registration, you can use the `OTDeleteName` function to delete a registered name.

If the mapper is in synchronous mode, the function does not return until the operation is complete. If the mapper is in asynchronous mode, the `OTDeleteName` function returns immediately. When the operation completes execution, the provider issues the `T_DELNAMECOMPLETE` event code. The `cookie` parameter to the notification routine has the value of the `name` parameter passed to the function. For more information on notifier functions and event codes see “Using Notifier Functions”(page 405) and “Event Codes”(page 383).

The `OTRegisterName` function (page 554) 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 (page 557) to delete a name using this ID value than to use the `OTDeleteName` function.

OTDeleteNameByID

Removes a previously registered name as specified by its name ID.

C INTERFACE

```
OSStatus OTDeleteNameByID (MapperRef ref,
                           OTNameID nameID);
```

C++ INTERFACE

```
TMapper::DeleteName(OTNameID nameID);
```

PARAMETERS

`ref`	A mapper reference.
`id`	The name ID, a 32-bit value specifying a number that identifies the registered name.
function result	An error code. See Appendix B (page 785)for more information.

DISCUSSION

If the mapper protocol defined using the `config` parameter to the `OTOpenMapper` or `OTAsyncOpenMapper` functions supports dynamic name and address registration, you can use the `OTDeleteNameByID` function to delete a registered name.

If the mapper is in synchronous mode, the function does not return until the operation is complete. If the mapper is in asynchronous mode, the `OTDeleteNameByID` function returns immediately. When the operation completes execution, the mapper provider calls the notifier function, passing `T_DELNAMECOMPLETE` for the `code` parameter, and the `nameID` parameter in the `cookie` parameter. For more information on notifier functions and event codes see “Using Notifier Functions”(page 405) and “Event Codes”(page 383).

The name ID that you delete using the `OTDeleteNameByID` function is returned in the `reply` parameter to the `OTRegisterName` function (page 554).

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

```
OSStatus OTLookupName(MapperRef ref,
                      TLookupRequest* req,
                      TLookupReply* reply);
```

C++ INTERFACE

```
OSStatus TMapper::LookupName(TLookupRequest* req,
                             TLookupReply* reply);
```

PARAMETERS

`ref`	A mapper reference.
`req`	A `TLookupRequest` structure (page 547) 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 548) that specifies the size and location of a buffer to hold the names found, and returns the number of names found.
function result	An error code. See Appendix B(page 785) for more information.

DISCUSSION

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 operation completes. On completion, the `reply` parameter's `names` field points to the buffer where the matching entries are stored and the rspcount field specifies the number of matching entries.

For each registered name found, the OTLookupName function stores information in the buffer referenced by the names field of the reply parameter. See Figure 6-1 (page 154) for the format of this information.

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. A sample program that shows the use of mapper function is Listing 6-1 (page 156).

If the mapper is in synchronous mode, the function does not return until the operation is complete. If you call the OTLookupName function asynchronously, the mapper provider calls your notifier function passing one of two completion event codes for the code parameter (T_LKUPNAMERESULT or T_LKUPNAMECOMPLETE) and passing the reply parameter in the cookie parameter. The mapper provider sends the T_LKUPNAMERESULT event each time it stores a name in the reply buffer, and it sends the T_LKUPNAMECOMPLETE event 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 operation 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 ignore it if it isn’t a name you’re interested in. Then, from inside your notifier you set the reply->names.len field or the reply->rspcount field back to 0 (thus allowing the mapper provider to overwrite the original 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. For more information on notifier functions and event codes see “Using Notifier Functions”(page 405) and “Event Codes”(page 383).

The cookie parameter to the notifier contains the reply parameter to the OTLookupName function.

The format of the names and protocol addresses are specific to the underlying protocol. Consult the documentation supplied for your protocol for more information.

SEE ALSO

The OTDeleteName function (page 556).

The OTDeleteNameByID function, (page 557).

OTNextLookupBuffer Macro

Returns the address of the next lookup buffer.

```
#define OTNextLookupBuffer(buf)\
    ((TLookupBuffer*)   \
        ((char*)buf + ((offsetof(TLookupBuffer, fAddressBuffer) +
buf->fAddressLength + buf->fNameLength + 3) & ~3)))
```

This macro takes the address of a TLookupBuffer and returns the address of the next one.

Option Management Reference

Contents

CHAPTER 25

This chapter describes the constants, 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. 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 32 bit 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 571) 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; however, you do not have to specify a value for the l_linger field of the linger structure. (page 571).

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 32-bit integer 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 32-bit integer specifying the low-water mark for the receive buffer—that is, the 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 Open Transport running out of memory and disabling incoming data packets.

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 32-bit integer 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 32-bit integer specifying the low-water mark for the send buffer—that is, the 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 protocol type used by the endpoint. The option is supported by the RawIP endpoint. For additional information, see "TCP/IP Services Reference"(page 683).

Generic Options

Open Transport defines some generic options that you can use with any protocol that supports 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
    OPT_ALERTENABLE         = (OTXTIName)0x0606,
};
```

Constant descriptions

`OPT_CHECKSUM` A 32-bit 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 receiver discards the packet.

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.

`OPT_RETRYCNT` A 32-bit integer specifying the number of times a function can attempt packet delivery before returning with an error. A value of 0 means that the function should attempt packet delivery an infinite number of times.

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.

`OPT_INTERVAL` A 32-bit integer specifying the interval of time in milliseconds that should elapse between attempts to deliver a packet. The number of attempts is defined by the `OPT_RETRYCNT` option.

`OPT_ENABLEEOM` An 32-bit integer 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. A value of 0 clears the option; a value of 1 sets it.

This option is not association-related.

`OPT_SELFSEND` A 32-bit integer allowing you to send broadcast packets to yourself. A value of 0 clears the option; a value of 1 sets it.

OPT_SERVERSTATUS A string that sets the server's status. The maximum length is protocol dependent. For more information, consult the documentation for the protocol you are using.

OPT_KEEPALIVE A keepalive structure (page 572) that specifies whether "keep alive" 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.

OPT_ALERTENABLE Enables or disables protocol alerts.

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 572) 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,
        T_UNSPEC            = (~0 - 2),
        T_ALLOPT            = 0
    };
```

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.

T_UNSPEC 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. The opt.buf field of the ret parameter points to a buffer where negotiated values for each option are placed.

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 565).

The linger structure is defined by the t_linger data type.

```
struct t_linger
{
long l_onoff;
long l_linger;
};
typedef struct t_linger t_linger;
```

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 567).

The keepalive structure is defined by the t_kpalive data type.

```
struct t_kpalive
{
    long kp_onoff;
    long kp_timeout;
};
typedef struct t_keepalive t_keepalive;
```

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 how many minutes Open Transport should 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, the protocol level of the option, the option name, the status 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;
    OTXTILevel      level;
    OTXTIName       name;
    UInt32          status;
    UInt32          value[1];
};
typedef struct TOption TOption;
```

Field descriptions

`len` The size (in bytes) of the option information, including the header.

`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 569).

`value` The option value. To have the endpoint select an appropriate value, you can specify the constant `T_UNSPEC`.

The TOption Header Structure

The `TOption` header structure stores information about options in a buffer.

```
struct TOptionHeader
{
    UInt32      len;
    OTXTILevel  level;
    OTXTIName   name;
    UInt32      status;
};

typedef struct TOptionHeaderTOptionHeader;

typedef struct TOption TOption;
```

Field descriptions

`len` The size (in bytes) of the option information, including the header.

`level` The protocol affected.

name The option name.

status The status value. Possible values are given by the status codes enumeration, (page 569)

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;
};

typedef struct TOptMgmt TOptMgmt
```

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 570). 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 569).

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 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

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

C++ INTERFACE

```
OSStatus 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 574), which describes the action to be taken by the function and the options affected.
`ret`	A pointer to an option management structure (page 574), which describes the options that were changed or returned by the function and how successful the negotiation process was.
function result	See Discussion.

DISCUSSION

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 synchronous mode, the endpoint will returns results of the option managment in the structure referenced by the `req` parameter. If the endpoint is in asynchronous mode, the function returns immediately with the `kOTNoError` result. When the operation completes, the endpoint calls your notifier with the `T_OPTNMGMTCOMPLETE` event code. The `cookie` parameter to the notifier will contain the value of the `ret` parameter. If you have not installed a notifier function, it is not possible to determine when the function completes. For more information on notifier functions and event codes see "Using Notifier Functions"(page 405) and "Event Codes"(page 383).

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 569).

  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(page 572). 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. (The option buffer should have an option header, but no value fields; the TOption.len field should be set to kOTOptionHeaderSize.)

  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.

  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, which you can get from the info.opt field of the TEndpointInfo(page 426) structure.

- 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.

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, which you can get from the info.opt field of the TEndpointInfo structure.

- 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. (The option buffer should have an option header, but no value fields; the TOption.len field should be set to kOTOptionHeaderSize.) 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 request for option negotiation is returned in the ret->flags field. The meaning of this result is described by the status codes enumeration (page 569).

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_OPTMGMTCOMPLETE` event before the function returns the first time.

SEE ALSO

The format of option buffers in “Specifying Option Values” (page 173).

The `OTCreateOptions` function (page 582).

The `OTCreateOptionString` function (page 582).

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

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 options.

`buflen` The size of the buffer containing the options.

`level` The protocol of the option to be found.

`name` The name of the option to be found.

function result The beginning address of the option found, or `nil` if it is not found.

DISCUSSION

Given a buffer such as might be returned by the `OTOptionManagement` function, you can use the `OTFindOption` function to find a specific option in the buffer.

SEE ALSO

The `OTNextOption` function (described next).

The `OTCreateOptionString` function (page 585).

OTNextOption

Locates the next `TOption` structure in a buffer.

C INTERFACE

```
OSStatus 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. On return, this parameter references the beginning address of the next option
function result	The only error returned by this function is `kOTBufferOverflow`; the option retrieved is larger than the buffer allocated to contain it.

DISCUSSION

The `OTNextOption` function allows you to parse through a buffer containing `TOption` structures(page 572) 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 `nil`. When the function returns, the `prevOptPtr` parameter points to the first 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 the next option in the buffer. You know the function has returned all existing `TOption` structures when `nextOption` is `nil`.

For an example of the use of this function, see "Sample Code: Getting and Setting Options" (page 177).

SPECIAL CONSIDERATIONS

Open Transport also defines a macro, `OPT_NEXTHDR`, that works like `OTNExtOption`, with less error checking.

SEE ALSO

The `OTFindOption` function (page 579).

Manipulating the Format of Option Information

You use the Open Transport utility functions described in this section to construct a buffer from a string describing option values 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 “Getting Started With Open Transport” (page 31).

OTCreateOptions

Writes option information into a buffer, from a string specifying option values.

C INTERFACE

```
OSStatus OTCreateOptions (const char* endptName, char** strPtr,
                          TNetbuf* buf);
```

C++ INTERFACE

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

PARAMETERS

`endptName` The name of the protocol for which the options are specified. 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.

`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 size of the buffer. To handle all possible options, create a buffer whose size is equal to the value specified by the `TEndpoint.options` field for an endpoint configured using the `endptName` string..

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, including padding.

function result See Discussion.

DISCUSSION

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` parameter. 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 (page 575).

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 next table describes how each value is represented.

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” (page 169).

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.

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.

SPECIAL CONSIDERATIONS

The option names used by this function are not localised. Thus, you should never present these option strings to users.

SEE ALSO

The `OTCreateOptionString` function (page 585).

OTCreateOptionString

Creates a string from a buffer containing `TOption` structures.

C INTERFACE

```
OSStatus OTCreateOptionString (const char* endptName, TOption** opt,
                    void* bufEnd, char* string,
                    size_t stringSize);
```

C++ INTERFACE

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

PARAMETERS

`endptName`	A constant specifying the name of the protocol for this option or options.
`opt`	A pointer to a pointer to a buffer containing one or more `TOption` structures(page 572).
`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.
function result	See Appendix B.

DISCUSSION

You can use the `OTCreateOptionString` function to parse through the options buffer returned by the `ret` parameter to the `OTOptionMangement` function (page 575) and create a string specifying option values that you can display.

SPECIAL CONSIDERATIONS

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

The `OTCreateOptions` function (page 582).

Ports Reference

Contents

This chapter describes the data types, constants, and functions that you need to obtain port information and register your application as a client. For conceptual information and code samples showing the use of these functions, see "Ports" (page 191).

Constants and Data Types

This section describes the constants and data types used in connection with ports.

Error-Checking Constant

Open Transport provides a constant that you can use to initialize the port reference structure. You can then check that the current port reference is valid before passing it to functions. The constant is defined as follows:

```
enum    kOTInvalidPortRef   = ((OTPortRef)0)
```

Port-Related Constants

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
    kMaxPortNameLength              = kMaxModuleNameLength + 4,
    kMaxPortNameSize                = kMaxPortNameLength + 1,
};
```

Bus Type Constants

You use the following constants to specify the bus type of port:

```
enum{
        kOTUnknownBusPort           = 0,
        kOTMotherboardBus           = 1,
        kOTNuBus                    = 2,
        kOTPCIBus                   = 3,
        kOTGeoPort                  = 4,
        kOTPCCordBus                = 5,
        kOTFireWireBus              = 6,
        kOTLastBusIndex             = 15
    };
```

Constant descriptions

`kOTUnknownBusPort`
The port's bus type is not a known type.

`kOTMotherboardBus`
The port is on the motherboard.

`kOTNuBus` The port is on a NuBus.

`kOTPCIBus` The port is on a PCI bus.

`kOTGeoPort` The port is a GeoPort device.

`kOTPCMCIABus` The port is on a PCCard bus.

`kOTFireWireBus` The port is on a Firewire bus.

`kOTLastBusIndex` The maximum bus type that the port can support.

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. See “OTRegisterAsClient” (page 614) for more information.

```
enum {
    kOTPortDisabled         = 0x25000001,
    kOTPortEnabled          = 0x25000002,
    kOTPortoffLine          = 0x25000003,
    kOTPortoffLine          = 0x25000004,
    kOTClosePortRequest     = 0x25000005,
    kOTYieldPortRequest     = 0x25000005,,
    kOTNewPortRegistered     = 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 specifies the reason, if known, and the `cookie` parameter provides the port reference of the port that went off line. A port going off line 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.

`kOTPortoffLine` The port is now offline.

`kOTClosePortRequest`A request has been made to close or yield this port.

`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 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 shareable, 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];
};
typedef struct OTPortRecord OTPortRecord;
```

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 .

fPortFlags
Flags describing the port's status. 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.

Flag	Value	Description
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.
kOTPortIsOffline	0x00000008	The port is off-line. This bit is typically only set when the port is active, the port autoconnects, and it is currently not connected.

fInfoFlags Flags providing additional information about the port.

Flag	Value	Description
kOTPortIsDLPI	0x00000001	The port driver is a DLPI STREAMS module.
kOTPortIsTPI	0x00000002	The port driver is a TPI STREAMS module.
kOTPortCanYield	0x00000004	The port can yield when requested.
kOTPortCanArbitrate	0x00000008	Reserved
kOTPortIsTransitory	0x00000010	The port has off-line/on-line status.
kOTPortAutoConnects	0x00000020	The port auto connects. The port goes on-line and off-line on demand. ISDN is a typical example.
kOTPortIsSystemRegistered	0x00004000	The port was registered by the system from the Name 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. This field is dependent on the ports device type. 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. When you get a Port Record, this pointer typically points into the SReserved field at the end of the record.

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; applications rarely need to 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 only 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 the 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 the 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 that contains a port's device and bus type, its slot number, and information to distinguish among several devices on a single slot. The actual structure of the port reference is private.

The port reference is defined by the `OTPortRef` data type.

```
typedef UInt32 OTPortRef;
```

You can use the `OTCreatePortRef` function (page 608) to create a port reference and obtain a pointer to it. The port reference is also a field in the port structure returned by the port information functions: `OTGetIndexedPort` (page 600), `OTFindPort` (page 601), and `OTFindPortByRef` (page 602). To extract information from the port reference, you need to use the functions `OTGetDeviceTypeFromPortRef` (page 604), `OTGetBusTypeFromPortRef` (page 605), and `OTGetSlotFromPortRef` (page 606).

This section lists the possible values for the device type and bus type.

IMPORTANT

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,
    kOTIrDADevice                   = 16,
    kOTATMSNAPDevice                = 17,
    kOTFibreChannelDevice           = 18,
    kOTFireWireDevice               = 19,
    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.

`kOTIrDADevice` The port is specified as an IrDA Infrared device.

`kOTATMSNAPDevice` The port is specified as an ATM pseudodevice simulating a SNAP device.

`kOTFibreChannelDevice`
The port is specified as a Fibre Channel device.

`kOTFireWireDevice` The port is specified as a Firewire device.

`kOTPseudoDevice` The port is designated as a pseudodevice.

`kOTLastDeviceIndex`
The maximum device types that a port can use.

`kOTLastSlotNumber`
The highest physical slot number a port can use. See "OTCreatePortRef" (page 608) for more information.

`kOTLastOtherNumber`
The maximum number of ports a single slot can support. See "OTCreatePortRef" (page 608) for more information.

Possible bus types are described in"Bus Type Constants" (page 590).

The Client List Structure

When you issue a yield port request with the `OTYieldPortRequest` function"OTYieldPortRequest" (page 611), 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 packed 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(page 611) 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. You should check the `kOTCanYieldPort` bit in the port structure’s `fInfoFlags` field to see whether the port supports yielding.

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` callin order to let the port go.

Functions

This section describes the functions you use to get information about ports, to obtain the use of ports from other owners, and to register as an Open Transport client.

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++ INTERFACE

```
OTPortRef TProvider::GetOTPortRef();
```

PARAMETERS

`ref`	The provider for which you wish to obtain the port.
function result	See Discussion.

DISCUSSION

The `OTGetProviderPortRef` function returns the port reference for the provider identified by the `ref` parameter. If the function returns `kOTInvalidPortRef`, 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++ INTERFACE

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

PARAMETERS

`record`	A pointer to a port structure (page 592) that contains, on return, information about a specific port on your computer.
`index`	An index number.
function result	See Discussion.

DISCUSSION

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 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

.

OTFindPort

Obtains information about a port that corresponds to a given port name.

C INTERFACE

```
Boolean OTFindPort  (OTPortRecord* record,
                     const char* portName);
```

C++ INTERFACE

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

PARAMETERS

`record`	A pointer to a port structure (page 592) that contains, on return, information about the port you specified with the `portName` parameter.
`portName`	The name of the port about which you want information.
function result	See Discussion.

DISCUSSION

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 the `OTFindPortByRef` function or 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 named port does not exist and the contents of the structure are not significant.

You can use the `OTGetIndexedPort` function (page 600) to get port information by iterating through all available ports.

SEE ALSO

"Obtaining Port Information" (page 195)

OTFindPortByRef

Obtains information about a port identified by its port reference.

C INTERFACE

```
Boolean OTFindPortByRef(OTPortRecord* record,
                        OTPortRef ref);
```

C++ INTERFACE

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

PARAMETERS

`record`	A pointer to a port structure (page 592) that contains, on return, information about the port you specified with the `ref` parameter.
`ref`	The port reference of the port about which you want information.
*function resul*t	See Discussion.

DISCUSSION

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.

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, no port exists with the given port reference and the contents of the structure are not significant.

SEE ALSO

“Obtaining Port Information” (page 195).

The OTCreatePortRef function (page 608).

OTGetUserPortNameFromPortRef

Returns a user-friendly name for a port.

C INTERFACE

```
void OTGetUserPortNameFromPortRef(OTPortRef ref,
                                  Str255 friendlyName);
```

C++ INTERFACE

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

PARAMETERS

ref	A port reference
friendlyName	A string specifying the port’s name.

DISCUSSION

For sample code showing the use of this function, see

SEE ALSO

“Obtaining Port Information” (page 195)

The `OTCreatePortRef` function (page 608).

OTGetPortIconFromPortRef

Returns the location for the icon family representing the port.

C INTERFACE

```
Boolean OTGetPortIconFromPortRef(OTPortRef ref,
                        OTResourceLocator* iconLocation);
```

C++ INTERFACE

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

PARAMETERS

`ref`	A port reference for the port whose icon you want to locate.
`iconLocation`	The location of the port's icon if it has one

SEE ALSO

"Obtaining Port Information" (page 195)

The `OTCreatePortRef` function (page 608).

OTGetDeviceTypeFromPortRef

Extracts the value of the hardware device type from a port reference.

C INTERFACE

```
UInt16 OTGetDeviceTypeFromPortRef(OTPortRef ref);
```

C++ INTERFACE

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

PARAMETERS

`ref`	The port reference from which you wish to extract the device type.
function result	Possible return values are listed in "The Port Reference" (page 595).

DISCUSSION

The `OTGetDeviceTypeFromPortRef` function extracts the device type value from a port reference. You can obtain such a port reference using the `OTGetIndexedPort` function (page 600) to access the port registry.

You can also use the `OTGetBusTypeFromPortRef` function (page 605) and the `OTGetSlotFromPortRef` function (page 606) to get the bus type and slot number information from the port reference.

SEE ALSO

"Obtaining Port Information" (page 195).

The `OTCreatePortRef` function (page 608).

OTGetBusTypeFromPortRef

Extracts the value of the bus type from a port reference.

C INTERFACE

```
UInt16 OTGetBusTypeFromPortRef(OTPortRef ref);
```

C++ INTERFACE

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

PARAMETERS

`ref`	The port reference from which you wish to extract the bus type.
function result	Possible return values are listed in "The Port Reference" (page 595)

DISCUSSION

The `OTGetBusTypeFromPortRef` function extracts the bus type value from a port reference. You can obtain such a port reference using the `OTGetIndexedPort` function (page 600) to access the port registry.

You can use the `OTGetDeviceTypeFromPortRef` function (page 604) and the `OTGetSlotFromPortRef` function (page 606) to get device type and slot number information from the port reference.

SEE ALSO

The `OTCreatePortRef` function (page 608).

"Obtaining Port Information" (page 195).

OTGetSlotFromPortRef

Extracts slot information from a port reference.

C INTERFACE

```
UInt16 OTGetSlotFromPortRef(OTPortRef ref,
                          UInt16* other);
```

C++ INTERFACE

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

PARAMETERS

`ref`	The port reference (page 595) from which you wish to extract the slot number.
`other`	A pointer to a 16-bit integer 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 (also called the multi-port identifier). Specify `NULL` for this parameter if you do not want the function to return this information.
function result	Possible return values are listed in “The Port Reference” (page 595).

DISCUSSION

The `OTGetSlotFromPortRef` function extracts slot information from a port reference. You can obtain such a port reference using the `OTGetIndexedPort` function (page 600) to access the port registry.

You can use the `OTGetDeviceTypeFromPortRef` function (page 604) and the `OTGetBusTypeFromPortRef` function (page 605) to get device type and bus type information from the port reference.

SPECIAL CONSIDERATIONS

In Open Transport port references, the slot numbers are physical slot numbers; 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

“Obtaining Port Information” (page 195).

The `OTCreatePortRef` function (page 608).

Registering New Ports

This section describes the functions used to register ports, create a port reference, and unregister ports.

OTRegisterPort

Register a port.

C INTERFACE

```
OSStatus OTRegisterPort(OTPortRecord* portInfo,
                        void* ref);
```

C++ INTERFACE

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

PARAMETERS

`portInfo`	A pointer to the port record for the port you want to register.
`ref`	A context pointer for your use.
function result	See Appendix B.

DISCUSSION

The function registers the port and returns the name it is registered under in the `fPortName` field of the port record.

Use the `OTUnregisterPort` function to unregister the port.

OTCreatePortRef

Creates a port reference that describes a unique port.

C INTERFACE

```
OTPortRef OTCreatePortRef(UInt8 busType,
                          UInt16 devType,
                          UInt16 slot,
                          UInt16 other);
```

C++ INTERFACE

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 589) 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 589) 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.
function result	A 32-bit port reference value.

DISCUSSION

The `OTCreatePortRef` function creates a port reference structure, which is a 32-bit value that describes a port: its device and bus type, its physical slot number, and, where applicable, its multiport identifier. 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.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 PCCard devices. 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 OTCreatePCCard PortRef(devType, slot, other)\
        OTCreatePortRef(kOTPCCardBus, 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` function (page 604),the `OTGetBusTypeFromPortRef` function (page 605), and the `OTGetSlotFromPortRef` function (page 606).

SPECIAL CONSIDERATIONS

In Open Transport port references the slot numbers are physical slot numbers; 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

“Obtaining Port Information” (page 195).

OTUnregisterPort

Unregisters a port.

C INTERFACE

```
OSStatus OTUnregisterPort(const char* portName,
                          void**);
```

C++ INTERFACE

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

PARAMETERS

`portName`	The name of the port you want to unregister.
function result	See Appendix B.

DISCUSSION

If you re-register the port, it might get a different name. Use `OTChangePortState` function if that is not desirable. Because a single `OTPortRef` can be registered with several names, Open Transport needs to use the `portName` rather than the `OTPortRef` to disambiguate.

SEE ALSO

The `OTRegisterPort` function (page 608).

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 ref, OTPortRef port,
                            OTClientList* buffer, size_t bufferSize);
```

C++ INTERFACE

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

PARAMETERS

`ref`	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.
`port`	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.
`bufferSize`	The size of the buffer, including the `fNumClients` field in the client list structure (page 597).
function result	See Discussion.

DISCUSSION

The `OTYieldPortRequest` function requests the current owner of a port (normally a serial port or modem) to yield ownership of it.

Port yielding works in one of two ways. You get the first mechanism by calling `OTYieldPortRequest`, passing a value for `buffer` that points to a client list buffer. Open Transport then looks through all the clients on the system. If the client has any providers that use the port in question, Open Transport sends the client a `kOTClosePortRequest` event. The client either handles that request (by closing the provider), or returns an error that indicates why the request was denied. Any client that doesn't have a notifier (as set by `OTRegisterAsClient`) is assumed to allow the request.

If any client denies the request, the `OTYieldPortRequest` function fails. The function could also fail 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 client currently using the port is already connected (`kENOENTErr`). Otherwise, Open Transport attempts to get the port for you from the driver. If it succeeds, the port is ready for you to use. The port remains available for a driver-specified amount of time (typically about 10 seconds). If you don't bind or connect the port in that time, the port reverts back to its previous owner. If you do bind or connect to the port before the timer expires, you now own the port. When you unbind or disconnect, the port reverts back to its previous owner.

The second mechanism for port yielding comes into play when you set `buffer` to `nil`. In that case, Open Transport bypasses the check described above, and immediately attempts to grab the port. The call results depend solely on whether that succeeds or fails.

After getting a port, you have a number of choices:

1. You can open it, as described above.
2. You can decide not to open it, and ignore it. After a while, the driver will time out, and the port will revert back to the previous owner.
3. You can decide not to open it, and be a good citizen by switching the port back to the previous owner immediately. You do this by calling the function `OTYieldPortRequest` and passing `-1` for the `buffer` parameter.

SPECIAL CONSIDERATIONS

On PowerPC computers, the `OTYieldPortRequest` function(page 611) is available only to PowerPC-native clients; there is no mixed-mode glue for the function, so you must build a fat application in order to use it.

You can only use this function at system task time.

SEE ALSO

The `OTRegisterAsClient` function (page 614).

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++ INTERFACE

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.
function result	An error code. See Appendix B (page 785)for more information.

DISCUSSION

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.

To unregister yourself as an Open Transport client, use the `OTUnregisterAsClient` function (page 615).

SEE ALSO

"Registering as an Open Transport Client" (page 198).

OTUnregisterAsClient

Removes your application as a client of Open Transport.

C INTERFACE

```
OSStatus OTUnregisterAsClient(void)
```

PARAMETERS

function result An error code. See Appendix B(page 785) for more information.

C++ INTERFACE

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

"Registering as an Open Transport Client" (page 198).

The `OTRegisterAsClient` function (page 614).

Utilities Reference

Contents

CHAPTER 27

This chapter describes Open Transport utility functions and the data structures used by these functions. For information about the use of these functions, please see “Utilities” (page 203).

Constants and Data Types

This section describes the data types and macros you need to use Open Transport utility functions.

The Timestamp Data Type

The timestamp data type is a 64-bit value that contains an Open Transport timestamp. The timestamp has unspecified units; you must use one of the timestamp manipulation functions described in “Timestamp Utility Functions” (page 635) to convert the timestamp to known quantities. The timestamp data type is defined by the `OTTimeStamp` data type.

```
typedef UnsignedWide OTTimeStamp;
```

▲ **WARNING**

The `OTTimeStamp` value is different in 68000-code and PowerPC code, so you cannot mix timestamp values obtained from emulated code with values obtained from PowerPC native code. ▲

The Lock Data Type

The lock data type defines a value that is used by the `OTClearLock` function(page 668) and the `OTAcquireLock` function(page 667) to ensure that Open Transport does not recursively reenter locked areas of code. The lock data type is defined by the `OTLock` data type(page 621).

```
typedef UInt8 OTLock;
```

The Linked List Structure

All of Open Transport's list utilities use the linked list structure, which may be embedded in any data structure that you want to use in an Open Transport list. A linked list structure is defined by the `OTLink` data type.

```
struct OTLink {
    OTLink* fNext;
};
typedef struct OTLink OTLink;
```

Field descriptions

`fNext` A pointer to the next entry in the linked list.

The LIFO List Structure

Open Transport LIFO (last-in, first-out) lists use the LIFO list structure. You must initialize this structure by setting the structure's `fHead` field to `NULL` before using the LIFO list. Most Open Transport LIFO list operations are atomic.

The LIFO list structure is defined by the `OTLIFO` data type.

```
struct OTLIFO {
    OTLink* fHead;
};
typedef struct OTLIFO OTLIFO;
```

Field descriptions

`fHead` A pointer to the first entry in the linked list. Set this to `nil` to initialize the structure before using it.

The FIFO List Structure

Open Transport FIFO (first-in, first-out) lists use the FIFO list structure. You must initialize this structure by setting the structure's `fHead` field to `NULL` before using the LIFO list. The FIFO list structure is defined by the `OTList` data type.

```
struct OTList {
    OTLink* fHead;
};
typedef struct OTList OTList;
```

Field descriptions

`fHead` A pointer to the first entry in the linked list. Set this to `NULL` to initialize the structure before using it.

▲ **WARNING**
None of the functions that handle a FIFO list structure are atomic. ▲

The Get Link Object Macro

Open Transport defines a macro (`OTGetLinkObject`) that you can use to easily cast back to the original object from a pointer to either a LIFO or a FIFO linked list structure.

```
DataType* OTGetLinkObject(OTLink* linkPtr,
                     DataType structName,
                     NameOfLinkPtrField fieldName)

#define OTGetLinkObject(link, struc, field)\
              ((struc*)((char*)(link) - offsetof(struc, field)))
```

PARAMETERS

`linkPtr` A pointer to the `OTLink` structure from which you wish to cast back.

`structName` The name of the structure containing the `OTLink` structure.

`fieldName` The `OTLink` field in the above structure.

function result The macro results in a pointer to the appropriate data type by offsetting the `linkPtr` appropriately and casting the result.

The Application-Defined List Search Function Prototype

In order to use the OTFindLink function(page 652) and the OTFindAndRemoveLink function(page 653), you must provide a user-defined function that is used when searching a list for a specific entry. The prototype for this function is as follows:

```
typedef Boolean (*OTListSearchProcPtr)
                        (const void* ref,
                        OTLink* linkToCheck);
```

For information about creating your own list search function, see MyListSearchFunction(page 669).

DISCUSSION

For 68000 code, this routine uses MPW C calling conventions. You should check your development environment documentation for information about defining a function with MPW C calling conventions. This is not an issue for PowerPC code.

Functions

This section describes the functions you use

- to allocate, free, and manipulate memory
- to manipulate strings
- to measure time
- to use lists
- to use atomic operations

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++ INTERFACE

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

PARAMETERS

`nbytes`	The amount (in bytes) of memory to allocate.

DISCUSSION

The `OTAllocMem` function allocates raw memory from a pool that Open Transport creates for your application. This function returns a pointer to the allocated memory; pass the same pointer to the `OTFreeMem` function to deallocate this memory.

SPECIAL CONSIDERATIONS

You can call this routine at both hardware interrupt level and from a deferred task. You need to call the `OTEnterInterrupt` function before you make this call from a hardware interrupt.

SEE ALSO

The `OTFreeMem` function (page 625).

OTFreeMem

Frees memory allocated from the Open Transport memory pool.

C INTERFACE

```
void OTFreeMem(void* arg)
```

C++ INTERFACE

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

PARAMETERS

`arg` A pointer to the memory to free. It must have been allocated by the `OTAllocMem` function.

DISCUSSION

The `OTFreeMem` function deallocates memory that you have allocated using the `OTAllocMem` function.

SPECIAL CONSIDERATIONS

You can call this routine at both hardware interrupt level and from a deferred task. You need to call the `OTEnterInterrupt` function before you make this call from a hardware interrupt.

SEE ALSO

The `OTAllocMem` function (page 625).

Memory Manipulation Utility Functions

Open Transport provides the following routines to allow you to copy memory, to compare the contents of memory, and to set memory.

OTMemcpy

Copies data from one memory location to another memory location.

C INTERFACE

```
void OTMemcpy       (void* dest,
                     const void* src,
                     size_t nBytes)
```

C++ INTERFACE

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

PARAMETERS

`dest`	A pointer to the location into which you are copying memory.
`src`	A pointer to the location from which you are copying memory.
`nBytes`	The number of bytes of memory you want to copy.

DISCUSSION

You can use this function for all memory operations except where

- the source or destination locations hold uncached memory. In these cases, the Mac OS function `BlockMoveDataUncached` is much faster.
- the source and destination areas overlap.

  Use the `OTMemmove` function (page 627) to copy data where the source and destination areas overlap. Otherwise use `OTMemcpy` because it is faster.

To compare the contents of two memory areas, use the `OTMemcmp` function (page 628).

OTMemmove

Copies data from one memory location to another memory location.

C INTERFACE

```
void OTMemmove      (void* dest,
                     const void* src,
                     size_t nBytes)
```

C++ INTERFACE

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

PARAMETERS

dest	A pointer to the location into which you are copying data.
src	A pointer to the location from which you are copying data.
nBytes	The number of bytes of data you want to copy.

DISCUSSION

You can use this function for all memory operations except where the source or destination locations hold uncached memory. In these cases, the Mac OS function `BlockMoveDataUncached` is much faster.

To move memory where the source and destination areas do not overlap, use the `OTMemcpy` function (page 626).

OTMemcmp

Compares the contents of two memory locations.

C INTERFACE

```
Boolean OTMemcmp    (constant void* mem1,
                     const void* mem2,
                     size_t nBytes)
```

C++ INTERFACE

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

PARAMETERS

`mem1`	A pointer to a memory location whose contents you are comparing with those at the location indicated by the `mem2` parameter.
`mem2`	A pointer to the memory location whose contents you are comparing with those at the location indicated by the `mem1` parameter.
`nBytes`	The number of bytes of memory you want to compare.
function result	The function returns a value of `true` if the indicated amount of memory at the two locations is the same.

OTMemzero

Initializes the specified memory range to 0.

C INTERFACE

```
void OTMemzero      (void* dest,
                     size_t nBytes)
```

C++ INTERFACE

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

PARAMETERS

`dest`	A pointer to a memory location whose contents you are zeroing out.
`nBytes`	The number of bytes of memory you want to zero out.

SEE ALSO

Use the `OTMemset` function (page 630) to set memory to a value other than 0.

OTMemset

Sets the specified memory range to a specific value.

C INTERFACE

```
void OTMemset       (void* dest,
                     uchar_p toSet,
                     size_t nBytes)
```

C++ INTERFACE

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

PARAMETERS

`dest`	A pointer to the memory location whose value you are setting.
`toSet`	The value you are setting the memory to.
`nBytes`	The number of bytes of memory you want to set.

DISCUSSION

Use the `OTMemzero` function (page 629) to set a specified memory range to 0.

Idling and Delaying Processing

Open Transport provides the following idle and delay processing functions.

OTIdle

Provides idle time to Open Transport.

C INTERFACE

```
void OTIdle(void)
```

C++ INTERFACE

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

DISCUSSION

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, it is provided for compatibility with existing programs that use an idling function.

SPECIAL CONSIDERATIONS

You cannot call the `OTIdle` function at primary interrupt time or at deferred task time. This function does not call the `SystemTask`, `WaitNextEvent`, or `GetNextEvent` functions.

IMPORTANT

You should never call the `OTIdle` function in production code on a Macintosh computer. ▲

OTDelay

Delays processing for a specified number of seconds.

C INTERFACE

```
void OTDelay(UInt32 seconds)
```

C++ INTERFACE

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

PARAMETERS

`seconds`	The number of seconds to delay.

DISCUSSION

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(page 631).

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.

IMPORTANT

You should never call the `OTIdle` function in production code on a Macintosh computer. ▲

String Manipulation Utility Functions

Open Transport provides functions that you can use to find out the length of a string, to copy a string, to concatenate two strings, and to determine whether two strings are the same.

OTStrLength

Returns the length of a C string.

C INTERFACE

```
size_t OTStrLength  (const char* string)
```

C++ INTERFACE

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

PARAMETERS

`string`	A zero-terminated string whose length you want to obtain.
function result	The length in bytes of the string.

OTStrCopy

Copies a C string.

C INTERFACE

```
void OTStrCopy      (char* dest,
                     const char* src)
```

C++ INTERFACE

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

PARAMETERS

`dest`	The memory location to which you want to copy the string indicated by the `src` parameter.

`src`	A zero-terminated string that you want to copy.

SEE ALSO

The `OTStrCat` function (page 634).

OTStrCat

Concatenates two C strings.

C INTERFACE

```
void OTStrCat       (char* dest,
                     const char* src)
```

C++ INTERFACE

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

PARAMETERS

`dest`	A zero-terminated string to which you want to concatenate the string specified by the `src` parameter.
`src`	A zero-terminated string that you want to concatenate to the string specified by the `dest` parameter.

OTStrEqual

Determines whether two C strings are the same.

C INTERFACE

```
Boolean OTStrEqual  (const char* str1,
                     const char* str2)
```

C++ INTERFACE

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

PARAMETERS

`str1`	A zero-terminated string that you want to compare with the string indicated by the `str2` parameter.
`str2`	A zero-terminated string that you want to compare with the string indicated by the `str1` parameter.
function result	The function returns a value of `true` when the two strings are exactly the same.

Timestamp Utility Functions

Open Transport provides the following functions to allow you to manipulate timestamp values.

▲ **WARNING**

The `OTTimeStamp` value is different in 68000 code and PowerPC code, so you cannot mix timestamp values obtained from emulated code with values obtained from PowerPC native code. ▲

OTGetTimeStamp

Obtains the current timestamp.

C INTERFACE

```
void OTGetTimeStamp (OTTimeStamp* stamp)
```

C++ INTERFACE

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

PARAMETERS

`stamp`	A pointer to the location where the function stores the current timestamp.

DISCUSSION

Use the `OTTimeStampInMilliseconds` function (page 637) to convert a timestamp value to milliseconds.

Use the `OTTimeStampInMicroseconds` function (page 638) to convert a timestamp value to microseconds.

Use the `OTSubtractTimeStamps` function (page 636) to obtain the difference between two timestamp values.

OTSubtractTimeStamps

Subtracts one timestamp value from another.

C INTERFACE

```
OTTimeStamp* OTSubtractTimeStamps(
                    OTTimeStamp* result,
                    OTTimeStamp* start,
                    OTTimeStamp* end)
```

C++ INTERFACE

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

PARAMETERS

`result`	On output, a pointer to the timestamp value that results from subtracting the value specified by the `start` parameter from the value specified by the `end` parameter.
`start`	A pointer to the timestamp value that you want to subtract.
`end`	A pointer to the timestamp value from which you want to subtract the value referenced by the `start` parameter.
function result	Returns the `result` parameter.

DISCUSSION

This function subtracts the timestamp referenced by the `start` parameter from the timestamp referenced by the `end` parameter. The return value is the same as the parameter `result`.

Use the `OTElapsedMilliseconds` function (page 639) to measure elapsed time in milliseconds.

Use the `OTElapsedMicroseconds` function (page 640) to measure elapsed time in microseconds.

OTTimeStampInMilliseconds

Converts the difference between two timestamp values into milliseconds.

C INTERFACE

```
UInt32 OTTimeStampInMilliseconds (OTTimeStamp* delta)
```

C++ INTERFACE

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

PARAMETERS

`delta` A reference to the value that you want to convert into milliseconds. You would normally get this value by calling `OTSubtractTimeStamps`(page 636).

function result The value specified by the `delta` parameter, converted into milliseconds.

DISCUSSION

A `delta` value of `0xffff ffff` indicates that the timestamp value has overflowed 32 bits of milliseconds.

Use the `OTGetTimeStamp` function (page 635) to obtain the current timestamp.

Use the `OTTimeStampInMicrosecondsFunction` (page 638) to convert a timestamp value to microseconds.

OTTimeStampInMicroseconds

Converts the difference between two timestamp values into microseconds.

C INTERFACE

```
UInt32 OTTimeStampInMicroseconds (OTTimeStamp* delta)
```

C++ INTERFACE

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

PARAMETERS

`delta` A reference to the value that you want to convert into microseconds. You would normally get this value by calling `OTSubtractTimeStamps`.

function result The number of microseconds obtained after the timestamp value has been converted.

DISCUSSION

A return value of `0xffff ffff` indicates that the converted `delta` value has overflowed 32 bits of microseconds.

Use the `OTGetTimeStamp` function (page 635) to obtain the current timestamp.

Use the `OTTimeStampInMilliseconds` (page 637) to convert a timestamp value to milliseconds.

OTElapsedMilliseconds

Calculates the time elapsed in milliseconds since the time specified by the `startTime` parameter.

C INTERFACE

```
UInt32 OTElapsedMilliseconds (OTTimeStamp* startTime)
```

C++ INTERFACE

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

PARAMETERS

`startTime`	A reference to a timestamp value returned by the `OTGetTimeStamp` function; this indicates the start of the elapsed time interval.
function result	The number of milliseconds that have elapsed since the timestamp specified by the `startTime` parameter.

DISCUSSION

Returns the number of milliseconds that have elapsed since the `OTGetTimeStamp` function was called to get the `startTime` value.

A return value of `0xffff ffff` indicates that the value has overflowed 32 bits of milliseconds.

Use the `OTElapsedMicroseconds` function (page 640) to measure elapsed time in microseconds.

Use the `OTGetClockTimeInSecs` function (page 641) to determine the number of seconds that have elapsed since system boot time.

OTElapsedMicroseconds

Calculates the time elapsed in microseconds since since the time specified by the `startTime` parameter.

C INTERFACE

```
UInt32 OTElapsedMicroseconds (OTTimeStamp* startTime)
```

C++ INTERFACE

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

PARAMETERS

`startTime`	A reference to a timestamp value returned by the `OTGetTimeStamp` function; this indicates the start of the elapsed time interval.
function result	The number of microseconds that have elapsed since the timestamp specified by the `startTime` parameter.

DISCUSSION

Returns the number of microseconds that have elapsed since the `OTGetTimeStamp` function was called to get the `startTime` value.

A return value of `0xffff ffff` indicates that the value has overflowed 32 bits of microseconds.

Use the `OTElapsedMilliseconds` function (page 639) to measure elapsed time in milliseconds.

Use the `OTGetClockTimeInSecs` function (page 641) to determine the number of seconds that have elapsed since system boot time.

OTGetClockTimeInSecs

Returns the number of seconds that have elapsed since system boot time.

C INTERFACE

```
UInt32 OTGetClockTimeInSecs (void)
```

C++ INTERFACE

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

PARAMETERS

function result Returns the number of seconds since the system was booted.

OTLIFO List Utility Functions

Open Transport provides several functions for dealing with an atomic LIFO (last-in, first-out) list. For code samples illustrating the use of these functions, see "Using List Management Functions" (page 204).

OTLIFOEnqueue

Places a link at the front of a LIFO list.

C INTERFACE

```
void OTLIFOEnqueue  (OTLIFO* list,
                     OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the list structure (page 622) in which the entry specified by the `link` parameter is to be placed.
`link`	A pointer to the `link` (page 622) being placed in the list.

DISCUSSION

This function atomically queues the entry referenced by the `link` parameter at the front of the list referenced by the `list` parameter.

Use the `OTLIFODequeue` function (page 642) to remove a link from a LIFO list.

OTLIFODequeue

Removes the first link in a LIFO list and returns a pointer to it.

C INTERFACE

```
OTLink* OTLIFODequeue (OTLIFO* list)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the LIFO list (page 622) from which you want to remove the link.
function result	A pointer to the link that has be removed.

DISCUSSION

This function atomically removes the first link in the specified LIFO list and returns a pointer to it. If there are no elements in the list, it returns `nil`.

Use the `OTLIFOStealList` function (page 643) to remove all links from a LIFO list.

OTLIFOStealList

Removes all links in a LIFO list and returns a pointer to the first link in the list.

C INTERFACE

```
OTLink* OTLIFOStealList (OTLIFO* list)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the LIFO list (page 622) from which you want to remove all linked entries.
function result	A pointer to the first link in the list.

DISCUSSION

This function atomically removes all of the elements from the specified LIFO list and returns a pointer to the first link in the list. You can access the other elements in the list by referring to the `fNext` field of the `OTLink` structures that make up the list.

If the list is empty, it returns `nil`.

OTReverseList

Reverses the order in which entries are linked in a list.

C INTERFACE

```
OTLink* OTReverseList (OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

`link`	A link that points to the linked list (page 622) whose elements you want to reverse; typically, this is the return value from the function `OTLIFOStealList`.
function result	A pointer to the first entry in the list referenced by the `link` parameter after the list's elements have been reordered.

DISCUSSION

This function does not reverse the list atomically. However, it is intended to be used after using the `OTLIFOStealList` function, to convert the list produced by that function into a usable FIFO list. A typical usage would be:

```
OTLink* link;
OTLink* next;
while ( (link = OTLIFOStealList(&myLifo)) != NULL )
{
    link = OTReverseList(link);
    do
    {
        MyObject* temp = OTGetLinkObject(link, MyObject, fMyLink);
        next = link->fNext;
        /* Do some stuff with temp */
    } while ( (link = next) != NULL );
}
```

This allows a producing task to be placing elements into a LIFO list, while a consuming task atomically processes them in FIFO order.

If link is `nil`, `OTReverseList` returns `nil`.

OTFIFO List Utility Functions

Open Transport provides several functions for dealing with an FIFO (first-in, first-out) list. None of the FIFO functions are atomic.

OTAddFirst

Places a link at the front of a FIFO list.

C INTERFACE

```
void OTAddFirst     (OTList* list,
                     OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the list structure (page 622) in which the entry specified by the link parameter is to be placed.
`link`	A pointer to the link (page 622) being placed in the list.

DISCUSSION

This function adds the link referenced by the `link` parameter to the front of the FIFO list referenced by the `list` parameter. Note that because it's a singly linked list, the `OTAddFirst` function is a lot faster than the `OTAddLast` function, expecially for long lists.

Use the `OTAddLast` function (page 646) to place a link at the end of a FIFO list.

Use the `OTRemoveFirst` function (page 647) to remove the first link in a FIFO list.

Use the `OTGetFirst` function (page 648) to obtain a pointer to the first link in a FIFO list.

OTAddLast

Adds a link to the end of a FIFO list.

C INTERFACE

```
void OTAddLast      (OTList* list,
                     OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the linked list (page 622) to which you want to add the link.
`link`	A pointer to the link (page 622) to be placed in the list.

DISCUSSION

This function adds the link referenced by the `link` parameter to the end of the FIFO list referenced by the `list` parameter.

Use the `OTAddFirst` function (page 645) to place a link at the front of a FIFO list.

Use the `OTRemoveLast` function (page 647) to remove the last link in a FIFO list.

Use the `OTGetLast` function (page 649) to obtain a pointer to the last link in a FIFO list.

OTRemoveFirst

Removes the first link in a FIFO list.

C INTERFACE

```
OTLink* OTRemoveFirst (OTList* list)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the FIFO list (page 622) from which you want to remove the link.
function result	A pointer to the link (page 622) that has been removed.

DISCUSSION

This function removes the first link in the FIFO list referenced by the `list` parameter and returns a pointer to it. It returns `nil` if the list is empty.

Use the `OTAddFirst` function (page 645) to place a link at the front of a FIFO list.

Use the `OTGetFirst` function (page 648) to obtain a pointer to the first link in a FIFO list.

OTRemoveLast

Removes the last link in a FIFO list.

C INTERFACE

```
OTLink* OTRemoveLast(OTList* list)
```

C++ INTERFACE

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

PARAMETERS

`list` A pointer to the FIFO list (page 622) from which you want to remove the link.

function result A pointer to the link (page 622) that has been removed.

DISCUSSION

This function removes the last link in the FIFO list referenced by the `list` parameter and returns a pointer to it. If the list is empty, it returns `nil`.

Use the `OTAddLast` function (page 646) to place a link at the end of a FIFO list.

Use the `OTGetLast` function (page 649) to obtain a pointer to the last link in a FIFO list.

OTGetFirst

Returns a pointer to the first element in a FIFO list.

C INTERFACE

```
OTLink* OTGetFirst  (OTList* list)
```

C++ INTERFACE

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

PARAMETERS

`list` A pointer to a FIFO list (page 622).

function result A pointer to the first element in the list.

DISCUSSION

This function returns a pointer to the first element in the FIFO list referenced by the `list` parameter. It does not remove the element from the list. If the list is empty, it returns `nil`.

Use the `OTAddFirst` function (page 645) to place a link at the front of a FIFO list.

Use the `OTRemoveFirst` function (page 647) to remove the first link in a FIFO list.

Use the `OTGetLast` function (page 649) to obtain a pointer to the last link in a FIFO list.

OTGetLast

Returns the last element in a FIFO list.

C INTERFACE

```
OTLink* OTGetLast   (OTList* list)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to a FIFO list (page 622).
function result	A pointer to the last element in the list.

DISCUSSION

This function returns a pointer to the last element in the FIFO list referenced by the `list` parameter. It does not remove the lement form the list. It returns `nil` if the list is empty.

The `OTAddLast` function (page 646) places a link at the end of a FIFO list.

Use the `OTRemoveLast` function (page 647) to remove the last link in a FIFO list.

Use the OTGetFirst function (page 648) to obtain a pointer to the first element in a FIFO list.

OTIsInList

Determines whether the specified link is in the specified list.

C INTERFACE

```
Boolean OTIsInList  (OTList* list,
                     OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

list	A pointer to a FIFO list (page 622).
link	A pointer to a link structure (page 622).
function result	Returns true if the specified link is in the FIFO list.

DISCUSSION

The OTIsInList function (page 650) determines whether a link is in a FIFO list.

Use the OTRemoveLink function (page 651) to remove a link from a FIFO list.

Use the OTFindLink function (page 652) or the OTGetIndexedLink function (page 654) to find a link in a FIFO list.

Use the OTFindAndRemoveLink function (page 653) to find and remove a link in a FIFO list.

OTRemoveLink

Removes a link from a FIFO list.

C INTERFACE

```
Boolean OTRemoveLink(OTList* list,
                     OTLink* link)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the FIFO list (page 622) from which you want to remove a link.
`link`	A pointer to the link (page 622) that you want to remove.
function result	Returns `true` if the specified link is in the specified list.

DISCUSSION

This function returns `true` if the element referenced by the `link` parameter is in the list referenced by the `list` parameter, and it removes the link from the list if it is.

Use the `OTRemoveFirst` function (page 647) to remove the first link in a FIFO list.

Use the `OTRemoveLast` function (page 647) to remove the last link in a FIFO list.

Use the `OTIsInList` function (page 650) to determine whether a link is in a FIFO list.

Use the `OTFindLink` function (page 652) or the `OTGetIndexedLink` function (page 654) to find a link in a FIFO list.

Use the `OTFindAndRemoveLink` function (page 653) to find and remove a link in a FIFO list.

OTFindLink

Finds a link in a FIFO list and returns a pointer to it.

C INTERFACE

```
OTLink* OTFindLink  (OTList* list,
                     OTListSearchProcPtr proc,
                     const void* refPtr)
```

C++ INTERFACE

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

PARAMETERS

`list`	A pointer to the FIFO list (page 622) to be searched.
`proc`	A pointer to the user-defined procedure to be used in searching for the link. See `MyListSearchFunction` (page 669) for information on defining your own list search function.
`refPtr`	A pointer to a value that is passed to the user-defined search procedure and that is useful to that procedure in finding a link. This value might be the address of a field, the value of a field, or any other kind of data that would help identify the link being sought.
function result	A pointer to the link being sought. The value of this pointer is `NULL` if no link matching your search criteria is found.

DISCUSSION

This function walks the list (from head to tail) specified by the `list` parameter, repeatedly calling the search procedure specified by the `proc` parameter. Each time it calls the search procedure, it passes it the value specified in the `refPtr` parameter. The `OTFindLink` function returns a pointer to the first `OTLink` value for the your `proc` procedure returned `true`. A `NULL` is returned if your `proc` procedure never returned `true`.

Use the `OTIsInList` function (page 650) to determine whether a link is in a FIFO list.

Use the `OTRemoveLink` function (page 651) to remove a link from a FIFO list.

Use the `OTFindAndRemoveLink` function (page 653) to find and remove a link in a FIFO list.

Use the `OTGetIndexedLink` function (page 654) to find a link in a FIFO list based on its index in the list.

OTFindAndRemoveLink

Finds a link in a FIFO list and removes it.

C INTERFACE

```
OTLink* OTFindAndRemoveLink(
                    OTList* list,
                    OTListSearchProcPtr proc,
                    const void* refPtr)
```

C++ INTERFACE

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

PARAMETERS

`list` A pointer to the FIFO list (page 622) to be searched.

`proc` A pointer to the user-defined procedure to be used in searching for the link. See `MyListSearchFunction` (page 669) for information on defining your own link.

`refPtr` A pointer to a value that is passed to the user-defined search procedure and that is useful to that procedure in finding a link. This value might be the address of a field, the value of a field, or any other kind of data that would help identify the link being sought.

function result A pointer to the link that was found and removed. The value of this pointer is `NULL` if no link matching your search criteria is found.

DISCUSSION

This function behaves exactly the same as the `OTFindLink` function (page 652)except that if a link value is found, it is removed from the list.

Use the `OTIsInList` function (page 650) to determine whether a link is in a FIFO list.

Use the `OTRemoveLink` function (page 651) to remove a link from a FIFO list.

Use the `OTFindLink` function (page 652) to find a link in a FIFO list.

Use the `OTGetIndexedLink` function (page 654) to find a link in a FIFO list based on its index in the list.

OTGetIndexedLink

Returns a pointer to the link at a specified position in a FIFO list.

C INTERFACE

```
OTLink* OTGetIndexedLink(
                    OTList* list,
                    size_t index)
```

C++ INTERFACE

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

PARAMETERS

`list` A pointer to the FIFO list (page 622) containing the link.

`index` The index of the link sought. The head of the list is at index 0.

function result A pointer to the link at the position specified by the `index` parameter, or `NULL` if `index` specifies a position that lies beyond the end of the list.

DISCUSSION

Use the `OTIsInList` function (page 650) to determine whether a link is in a FIFO list.

Use the `OTFindLink` function (page 652) to find a link in a FIFO list.

Adding and Removing aList Element

Open Transport provides two functions you can use to add or remove and element from a list.

OTEnqueue

Adds an element to a list.

C INTERFACE

```
void OTEnqueue(void** listHead,
                      void* object,
                      size_t linkOffset);
```

C++ INTERFACE

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

PARAMETERS

`listHead` A pointer to the list head.

`object` The address of the object you want to insert in the list.

`linkOffset`	The size of the object. That is, the distance in bytes from the beginning of object to the `fNext` field in its linked list entry.

DISCUSSION

This function puts the address you pass in the `object` parameter in the list head and places the element that was previously referenced in the list head into the pointer at `object` plust `linkOffset`.

OTDequeue

Adds an element to a list.

C INTERFACE

```
void OTDequeue(void** listHead,
                     size_t linkOffset);
```

C++ INTERFACE

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

PARAMETERS

`listHead`	A pointer to the list head.
`linkOffset`	The size of the object. That is, the distance in bytes from the beginning of object to the `fNext` field in its linked list entry.

DISCUSSION

This functionreturns the address of the head object in the list and places the address of the next element in the list at the list head.

Atomic Operations

Open Transport supplies a number of atomic functions that you can use in your code. The primary reason for these routines is that atomic operations minimize the need for turning interrupts on and off. A secondary reason is to make your code multiprocessor safe.

OTAtomicSetBit

Sets a specified bit in a byte.

C INTERFACE

```
Boolean OTAtomicSetBit(
                        UInt8* bytePtr,
                        size_t bitToSet)
```

C++ INTERFACE

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

PARAMETERS

bytePtr	A pointer to the byte containing the bit to be set
bitToSet	A value ranging from 0 to 7 that specifies the bit to set.
function result	Returns the previous state of the bit: true if the bit was set; false, otherwise.

DISCUSSION

This function atomically sets the bit specified by the bitToSet parameter, in the byte referenced by the bytePtr parameter. This function returns the previous state of the bit.

No error checking is done on the bitToSet value you specify, and results are undefined if the value lies outside the range 0 through 7.

Use the OTAtomicClearBit function (page 658) to clear a bit value.

Use the OTAtomicTestBit function (page 659) to test a bit value.

OTAtomicClearBit

Clears a bit in a byte.

C INTERFACE

```
Boolean OTAtomicClearBit(
                        UInt8* bytePtr,
                        size_t bitToClear)
```

CC++ INTERFACE

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

PARAMETERS

bytePtr	A pointer to the byte containing the bit to clear.
bitToClear	A value ranging from 0 to 7 that specifies the bit to clear.
function result	Returns the previous state of the bit: true if the bit was set; false, otherwise.

DISCUSSION

This function atomically clears the bit specified by the bitToClear parameter, in the byte referenced by the bytePtr parameter. This function returns the previous state of the bit.

No error checking is done on the bitToClear value you specify, and results are undefined if the value lies outside the range 0 through 7.

Use the OTAtomicSetBit function (page 657) to set a bit value.

Use the OTAtomicTestBit function (page 659) to test a bit value.

OTAtomicTestBit

Tests a bit in a byte and returns its current state.

C INTERFACE

```
Boolean OTAtomicTestBit(
                UInt8* bytePtr,
                size_t bitToTest)
```

C++ INTERFACE

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

PARAMETERS

`bytePtr`	A pointer to the byte containing the bit to test.
`bitToTest`	A value ranging from 0 to 7 that specifies the bit to test.
function result	Returns the current state of the bit: `true` if the bit is set; `false`, otherwise.

DISCUSSION

This function atomically tests the bit specified by the `bitToTest` parameter, in the byte referenced by the `bytePtr` parameter.

No error checking is done on the `bitToTest` value you specify, and results are undefined if the value lies outside the range 0 through 7.

Use the `OTAtomicSetBit` function (page 657) to set a bit value.

Use the `OTAtomicClearBit` function (page 658) to clear a bit value.

OTCompareAndSwapPtr

Atomically compares the value of a pointer at a memory location and atomically swaps it with a second pointer value if the compare is successful.

C INTERFACE

```
Boolean OTCompareAndSwapPtr(
                     void* oldVal,
                     void* newVal,
                     void** where)
```

C++ INTERFACE

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

PARAMETERS

`oldVal`	The pointer being compared to the pointer referenced by the `where` parameter.
`newVal`	The value to be assigned to the pointer referenced by the `where` parameter.
`where`	A reference to the pointer being compared to the pointer specified by the `oldVal` parameter.
function result	Returns `true` if the swap is done.

DISCUSSION

This function compares the value specified by the `oldVal` parameter with the value referenced by the `where` parameter. If the two values are the same, the function replaces the values referenced by the `where` parameter with the value specified by the `newVal` parameter and it returns `true`. If the two values are not the same, no swap occurs and the function returns `false`.

The entire compare and swap operation is atomic.

▲ **WARNING**

The pointer `where` must be on a 4-byte boundary, or calling this function may hang the machine. ▲

Use the `OTCompareAndSwap32` function (page 661) to compare two 32-bit values and set one if they are the same.

OTCompareAndSwap32

Atomically compares two 32-bit values and changes one of these values if they are the same.

C INTERFACE

```
Boolean OTCompareAndSwap32(
                      UInt32 oldVal,
                      UInt32 newVal,
                      UInt32* where)
```

C++ INTERFACE

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

PARAMETERS

`oldVal`	The 32-bit value being compared to the value referenced by the `where` parameter.
`newVal`	The value to be assigned to the long-word referenced by the `where` parameter.
`where`	A reference to the 32-bit value being compared to that specified by the `oldVal` parameter.
function result	Returns `true` if the swap is done.

DISCUSSION

This function compares the value specified by the `oldVal` parameter with the value referenced by the `where` parameter. If the two values are the same, the function replaces the long-word referenced by the `where` parameter with the value specified by the `newVal` parameter and it returns `true`. If the two values are not the same, no swap occurs and the function returns `false`.

The entire compare and swap operation is atomic.

▲ **WARNING**

The pointer `where` must be on a 4-byte boundary, or calling this function may hang the machine. ▲

Use the `OTCompareAndSwapPtr` function (page 659) to compare two pointer values and set one if they are the same.

Use the `OTCompareAndSwap16` function (page 662) to compare two 16-bit values and set one if they are the same.

Use the `OTCompareAndSwap8` function (page 663) to compare two 8-bit values and set one if they are the same.

OTCompareAndSwap16

Atomically compares two 16-bit values and changes one of these values if they are the same.

C INTERFACE

```
Boolean OTCompareAndSwap16(
                    UInt32 oldVal,
                    UInt32 newVal,
                    UInt16* where)
```

C++ INTERFACE

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

PARAMETERS

`oldVal` The 16-bit value being compared to the value referenced by the `where` parameter.

`newVal` The value to be assigned to the 16-bit value referenced by the `where` parameter.

`where` A reference to the 16-bit value being compared to that specified by the `oldVal` parameter.

function result Returns true if the swap is done.

DISCUSSION

This function compares the least significant 16 bits of oldVal with the value referenced by the where parameter. If the two values are the same, the function replaces the 16-bit value referenced by the where parameter with the value specified by the newVal parameter and it returns true. If the two values are not the same, no swap occurs and the function returns false.

The entire compare and swap operation is atomic.

▲ **WARNING**
The pointer where must be at an address where the second byte of the 16 bit value is not in a different 4-byte cell than the first byte (i.e. (where % 4) != 3). ▲

Use the OTCompareAndSwap32 function (page 661) to compare two 32-bit values and set one if they are the same.

Use the OTCompareAndSwap8 function (page 663) to compare two 8-bit values and set one if they are the same.

OTCompareAndSwap8

Atomically compares two 8-bit values and changes one of these values if they are the same.

C INTERFACE

```
Boolean OTCompareAndSwap8(
                    UInt32 oldVal,
                    UInt32 newVal,
                    UInt8* where)
```

C++ INTERFACE

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

PARAMETERS

`oldVal` The 8-bit value being compared to the value referenced by the `where` parameter.

`newVal` The value to be assigned to the 8-bit value referenced by the `where` parameter.

`where` A reference to the 8-bit value being compared to that specified by the `oldVal` parameter.

function result Returns `true` if the swap is done.

DISCUSSION

This function compares the least significant 8 bits of `oldVal` with the value referenced by the `where` parameter. If the two values are the same, the function replaces the 8-bit value referenced by the `where` parameter with the value specified by the `newVal` parameter and it returns `true`. If the two values are not the same, no swap occurs and the function returns `false`.

The entire compare and swap operation is atomic.

Use the `OTCompareAndSwap32` function (page 661) to compare two 32-bit values and set one if they are the same.

Use the `OTCompareAndSwap16` function (page 662) to compare two 16-bit values and set one if they are the same.

OTAtomicAdd32

Atomically adds a 32-bit value to a memory location.

C INTERFACE

```
SInt32 OTAtomicAdd32(SInt32 toAdd,
                     SInt32* where)
```

C++ INTERFACE

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

PARAMETERS

toAdd	A signed 32-bit value to add to the value referenced by the where parameter.
where	A pointer to a signed 32-bit value that is added to the value specified by the parameter toAdd.
function result	The sum of the values specified by the function's parameters.

DISCUSSION

This function atomically adds the signed 32-bit value specified by the parameter toAdd to the value referenced by the where parameter and returns the result. It also stores the result in the location referenced by the where parameter.

You can use this function to subtract from the toAdd value referenced by the where parameter by reversing the sign of toAdd.

▲ **WARNING**
The pointer where must be on a 4-byte boundary, or calling this function may hang the machine. ▲

Use the OTAtomicAdd16 function (page 665) to add two 16-bit values.

Use the OTAtomicAdd8 function (page 666) to add two 8-bit values.

OTAtomicAdd16

Atomically adds a 16-bit value to a memory location.

C INTERFACE

```
SInt16 OTAtomicAdd16(SInt32 toAdd,
                     SInt16* where)
```

C++ INTERFACE

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

PARAMETERS

`toAdd` A signed 16-bit value to add to the value referenced by the where `parameter`.

`where` A pointer to a signed 16-bit value that is added to the value specified by the parameter `toAdd`.

function result The sum of the values specified by the function's parameters.

DISCUSSION

This function atomically adds the least significat 16 bits of `toAdd` to the value referenced by the `where` parameter and returns the result. It also stores the result in the location referenced by the `where` parameter

You can use this function to subtract from the toAdd value from the value referenced by the `where` parameter by reversing the sign of `toAdd`.

▲ **WARNING**
The pointer `where` must be at an address where the second byte of the 16 bit value is not in a different 4-byte cell than the first byte (i.e. (`where % 4) != 3`). ▲

Use the `OTAtomicAdd32` function (page 664) to add two 32-bit values.

Use the `OTAtomicAdd8` function (page 666) to add two 8-bit values.

OTAtomicAdd8

Atomically adds an 8-bit value to a memory location.

C INTERFACE

```
SInt8 OTAtomicAdd8  (SInt32 toAdd,
                     SInt8* where)
```

C++ INTERFACE

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

PARAMETERS

`toAdd`	A signed 8-bit value to add to the value referenced by the where `parameter`.
`where`	A pointer to a signed 8-bit value that is added to the value specified by the parameter `toAdd`.
function result	The sum of the values specified by the function's parameters.

DISCUSSION

This function atomically adds the least significant 8 bits of `toAdd` to the value referenced by the `where` parameter and returns the result. It also stores the result in the location referenced by the `where` parameter.

You can use this function to subtract the toAdd value from the value referenced by the `where` parameter by reversing the sign of `toAdd`.

Use the `OTAtomicAdd32` function (page 664) to add two 32-bit values.

Use the `OTAtomicAdd16` function (page 665) to add two 16-bit values.

Locking Functions

Open Transport provides the following locking functions so that you can protect "critical sections" of your code. Protecting a critical section ensures that this section is accessed by at most one caller. You might want to create such a section if it can be accessed by more than one caller and you want to guarantee each caller exclusive access to that section.

OTAcquireLock

Acquires a lock and marks the beginning of a critical section.

C INTERFACE

```
Boolean OTAcquireLock (OTLock* lockPtr)
```

C++ INTERFACE

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

PARAMETERS

`lockPtr`	A pointer to a lock (page 621) used to identify this section.
function result	Returns `true` if the lock is acquired.

DISCUSSION

This function attempts to acquire the lock referenced by the `lockPtr` parameter. If successful, `true` is returned. If not, then some other thread is holding the lock, and `false` is returned.

Use the `OTClearLock` function (page 668) to clear a lock aquired with the `OTAcquireLock` function and to mark the end of a critical section.

OTClearLock

Clears a lock and marks the end of a critical section.

C INTERFACE

```
void OTClearLock    (OTLock* lockPtr)
```

C++ INTERFACE

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

PARAMETERS

`lockPtr`	A pointer to a lock (page 621) that you want to release.

DISCUSSION

This function clears the lock specified by the `lockPtr` parameter. You should only attempt to clear the lock if `OTAquireLock` returned `true`.

Use the `OTAcquireLock` function (page 667) to acquire a lock to a critical section.

Application-Defined Functions

This section describes the user-defined function that you need to provide in order to find an entry in a linked list.

MyProcessCallbackFunction

For information about creating and using a callback function see "MyOTProcessProc" (page 541).

MyListSearchFunction

Search through a list for a link that matches certain criteria.

```
Boolean (MyListSearchFunction)
                        (const void* ref,
                        OTLink* linkToCheck);
```

PARAMETERS

`ref` A value passed to your function by the `OTFindLink` or `OTFindAndRemoveLink` functions. This is the value that your function must match against each link in the list until it finds a matching link. It is the same as the refPtr parameter passed to `OTFindLink` and `OTFindAndRemoveLink`.

`linkToCheck` A pointer to the current link being evaluated.

function result The function should return `true` if the list element defined by `linkToCheck` matches the information specified by the `refPtr` parameter.

DISCUSSION

When the `OTFindLink` function(page 652) or the `OTFindAndRemoveLink` function(page 653) execute, they call this user-defined function.

For 680x0 code, this routine uses MPW C calling conventions. You should check your development environment documention for information about defining a function with MPW C calling conventions. This is not an issue for PowerPC code.

Advanced Topics Reference

Contents

This chapter provides reference information for sending non-contiguous data, doing no-copy receives, and using raw-data mode. For a discussion of how to use these data structures and functions in your program, see “Advanced Topics” (page 215).

Constants and Data Types

This section describes the constants and data types you use to specify non-contiguous data and to do no-copy receives.

OTData Constant

When sending data that is noncontiguous, you need to use an OTData buffer. Open Transport provides a constant, kNetbufDataIsOTData, that you can use for the TNetbuf.len field when you send data to indicate that the value in the TNetbuf.buf structure is actually a pointer to an OTData buffer.

```
enum {
    kNetbufDataIsOTData     = 0xfffffffe
};
```

OTBuffer Constant

When receiving data without making a copy, you need to pass an OTBuffer pointer as your data buffer 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= 0xfffffffd
};
```

Raw Mode Constants

This flag is used in the `udata.addr.len` field for `OTSndUData` to indicate the use of raw mode.

```
enum {
    kNetbufIsRawMode = 0xffffffff
}
```

The OTData Structure

You use the `OTData` structure to specify the location and size of noncontiguous data. You use a pointer to this structure in place of a pointer to continguous data normally referenced in `TNetbuf.buf` field. You can send discontiguous data using the `OTSndUData` function (page 462), the `OTSndURequest` function (page 469), the `OTSndUReply` function (page 475), the `OTSnd` function (page 494), the `OTSndRequest` function (page 499), and the `OTSndReply` function (page 504).

Note
The `OTData` structure is an Apple extension. ▲

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. For more information, see “Sending Noncontiguous Data” (page 216).

```
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` Specifies the size of the fragment in bytes.

The No-Copy Receive Buffer Structure

You use the no-copy receive buffer structure when you wish to receive data without copying it with the `OTRcvUData` function (page 467), the `OTRcvURequest` function (page 472), the `OTRcvUReply` function (page 478), the `OTRcv` function (page 496), the `OTRcvRequest` function (page 502), and the `OTRcvReply` function (page 507).

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 no-copy receive 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. ▲

For more information, see “No-Copy Receiving” (page 218).

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(page 676) (OTBuffer) is read-only, you may need to copy the data in sections as you progress through the no-copy receive buffer. The utility function OTReadBuffer(page 679) is used with this structure to easily copy the data out of an OTBuffer structure.

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 far into the buffer you have read.

`fPad` Reserved.

Functions

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 (page 675) to be released.

DISCUSSION

Once a no-copy receive is completed, you need to release the `OTBuffer` structure as quickly as possible by calling this function.

SPECIAL CONSIDERATIONS

This function is available only to PowerPC-native clients; there is no mixed-mode glue for the function, so you must build a fat application in order to use it.

OTBufferDataSize

Obtains the size of the no-copy receive buffer.

C INTERFACE

```
size_t OTBufferDataSize(OTBuffer* buf);
```

C++ INTERFACE

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

PARAMETERS

`buf`	A pointer to a no-copy receive buffer (page 675).
function result	The number of bytes in the no-copy receive buffer.

DISCUSSION

Use the `OTReadBuffer` (page 679) function to copy from the buffer referenced by the `buf` parameter.

Use the `OTReleaseBuffer` function (page 677)) to return the no-copy receive buffer to the system. .

SPECIAL CONSIDERATIONS

This function is available only to PowerPC-native clients; there is no mixed-mode glue for the function, so you must build a fat application in order to use it.

OTReadBuffer

Copies data out 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 (page 675) into which to copy the data.
`len`	The number of bytes to copy.
function result	See Discussion.

DISCUSSION

This function copies `len` bytes from offset `info->fOffset` of the no copy receive buffer pointed to by `info->fBuffer` to the buffer specified by `buf`. It returns `false` if there is more data to be copied, or `true` if the data is exhausted.

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.

Use the OTReleaseBuffer function (page 677) to return this structure to the system.

SEE ALSO

The OTBufferDataSize function (page 678).

TCP/IP Services Reference

Contents

This chapter documents the constants, data structures and functions used by the TCP/IP service provider. It also provides mapper and endpoint ifnormation for TCP/IP. The chapter "TCP/IP Services" (page 237) describes the use of these structures and functions in opening and using TCP/IP endpoints and mappers.

Constants and Data Types

This section describes the constants and data types used by the TCP/IP service interface.

Basic Types and Constants

You use the following 16 bit value to distinguish among ports.

```
typedef UInt16 InetPort;
```

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 kTCPName                "tcp"
#define kUDPName                "udp"
#define kRawIPName              "rawip"
#define kDNRName                "dnr"
```

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. For details, see "Internet Address Structure" (page 685) and "DNS Address Structure" (page 686)..

```
enum {
        AF_INET                 = 2,
        AF_DNS                  = 42
};
```

You can use the kOTAnyInetAddress value with the OTBind function when you are not concerned with the specific IP interface you are binding to.

```
enum {
        kOTAnyInetAddress       = 0
};

enum {
        kMaxHostAddrs           = 10,
        kMaxSysStringLen        = 32,
        kMaxHostNameLen         = 255
};
typedef char InetDomainName[kMaxHostNameLen];
```

kMaxHostAddrs The maximum number of hosts returned by the function OTInetStringToAddress.

kMaxSysStringLen The maximum length of the strings returned by the function OTInetSysInfo.

kMaxHostNameLen The maximum length of an Internet domain name.

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 sent by TCP/IP service provider functions:

```
enum {
        T_DNRSTRINGTOADDRCOMPLETE           = 0x10000001,
        T_DNRADDRTONAMECOMPLETE             = 0x10000002,
        T_DNRSYSINFOCOMPLETE                 = 0x10000003,
        T_DNRMAILEXCHANGECOMPLETE            = 0x10000004,
        T_DNRQUERYCOMPLETE                   = 0x10000005
};

enum {
        kDefaultInetInterface                = -1,
        kInetInterfaceInfoVersion            = 2
};
```

`kDefaultInetInterface`
: Value passed to `OTInetGetInterfaceInfo` (page 711) to get the interface configured in the TCP/IP control panel.

`kInetInterfaceInfoVersion`
: Version number of the correct `InetInterfaceInfo` structure.

The `SET_TOS` macro is used when negotiating IP_TOS option.

```
#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`(page 484), `OTSndURequest`(page 469), and `OTBind`(page 441).
You can use the `OTInitInetAddress` function (page 713) to fill in an internet address structure. The internet address structure is defined by the `InetAddress` data type.

```
struct InetAddress {
        OTAddressType       fAddressType;
        InetPort            fPort;
        InetHost            fHost;
        UInt8               fUnused[8];
};
typedef struct InetAddress InetAddress;
```

Field descriptions

`fAddressType` The address type. The field should be `AF_INET`, which identifies the structure as an `InetAddress`.

fPort	The port number.
fHost	The 32-bit IP address of the host.
fUnused	Reserved.

DNS Address Structure

You can use the DNS (domain name system) address structure with the OTConnect function(page 484) (TCP), with the OTSndUData function (UDP)(page 462), or with the OTResolveAddress function (either TCP or UDP)(page 453). If you do so, TCP/IP will resolve the name for you automatically. You can use the OTInitDNSAddress function (page 714) to fill in a DNS address structure. The DNS address structure is defined by the DNSAddress data type.

```
struct DNSAddress {
    OTAddressType      fAddressType;
    InetDomainName     fName;
};
typedef struct DNSAddress DNSAddress;
```

Field descriptions

fAddressType	The address type. For a DNSAddress structure, this should be AF_DNS.
fName	The name to be resolved by the DNR.

The 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 a 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.

DNS Query Information Structure

The DNS query information structure is used by the TCP/IP service provider to return answers to DNS queries made using the OTInetQuery function. The DNS query information structure is defined by the DNSQueryInfo data type. For additional information about the constant values for the DNSQueryInfo (page 687) fields, see the DNS Requests for Comments (RFCs), available over the World Wide Web.

```
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 numerical value of the DNS resource record type, such as MX and PTR, for which you wish to query.

qClass
: The numerical value of the DNS record class, such as Inet and Hesio, for which you wish to query.

ttl
: An integer indicating the DNS resource record's time to live (in seconds).

name
: The fully qualified domain name or address for which you made the query.

responseType
: The type of response.

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.

See the Internet Standard for a definitive list of values for the qType, qClass, and respnoseType fields, and for a definition of the format of the resource data.

Internet Interface Information Structure

The `OTInetGetInterfaceInfo` function (page 711) returns information about an interface on 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;
    InetHost            fNetmask;
    InetHost            fBroadcastAddr;
    InetHost            fDefaultGatewayAddr;
    InetHost            fDNSAddr
    UInt16              fVersion;
    UInt16              fHWAddrLen
    UInt8*              fHWAddr;
    UInt32              fIfMTU
    UInt8*              fReservedPtrs[2];
    InetDomainName      fDomainName;
    UInt32              fIPSecondaryCount;
    UInt8               fReserved[256];
};
typedef struct InetInterfaceInfo InetInterfaceInfo;
```

Field descriptions

`fAddress` The IP address of the interface.

`fNetMask` The subnet mask of the local IP network.

`fBroadcastAddr` The broadcast address for the interface.

`fDefaultGatewayAddr`
The IP address of the default router. The default is a router that can forward any packet destined outside the locally 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.

`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. If multiple DNS servers are available, only the first is returned.

fVersion The version of the OTInetGetInterfaceInfo function; currently equal to kInetInterfaceInfoVersion.

fHWAddrLen The length (in bytes) of the hardware address. This points into the fReserved field of this structure. It can be nil if the interface has no hardware address or if it won't fit.

fHWAddr A pointer to the hardware address.

fIfMTU The maximum transmission unit size in bytes permitted for this interface's hardware.

fReservedPtrs Reserved.

fDomainName The default domain name of the host as configured in the TCP/IP control panel. This name doesn't include the host name.

fIPSecondaryCount The number of IP secondary address available.

fReserved Reserved.

Internet Host Information Structure

The OTInetStringToAddress function (page 700) 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];
};
typedef struct InetHostInfo Inet HostInfo;
```

Field descriptions

name The **canonical name** of the host. The canonical name is a fully qualified domain nam, never 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 705) 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];
};
typedef struct InetSysInfo InetSysInfo;
```

Field descriptions

`cpuType` The CPU type of the specified host. This is an ASCII string maintained by the domain name server.

`osType` The operating system running on the specified host. This is an ASCII 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 699) 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;
};
typedef struct TIPAddMulticast TIPAddMulticast;
```

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 707) 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;
};
typedef struct InetMailExchange InetMailExchange;
```

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. For more detailed descriptions of how to use these options and what sizes they are, see *X/Open CAE Specification (1992): X/Open Transport Interface (XTI)*. For additional information about generic options and option use, see “Option Management” (page 165).

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,      /* IP */
    INET_TCP    = 0x06,     /* TCP */
    INET_UDP    = 0x11      /* UDP */
};
```

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
#define TCP_MAXSEG                           0x02
#define TCP_NOTIFY_THRESHOLD                 0x10
#define TCP_ABORT_THRESHOLD                  0x11
#define TCP_CONN_NOTIFY_THRESHOLD            0x12
#define TCP_CONN_ABORT_THRESHOLD             0x13
#define TCP_OOBINLINE                        0x14
#define TCP_URGENT_PTR_TYPE                  0x15
#define TCP_KEEPALIVE                OPT_KEEPALIVE
```

Option descriptions

TCP_NODELAY Sets the TCP delay mode. (UInt32)

By default, when TCP has a full segment's worth of outgoing data, it sends the segment immediately. However, if it receives less than a segment's worth of outgoing data and has not yet received acknowledgment for the last packet sent, it saves the data until it receives a full segment's worth, or it receives acknowledgment for the last packet, or 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. This means that TCP must support the option, not that the client must use it.

TCP_MAXSEG Read the maximum TCP segment size. (UInt32)

The maximum segment size is returned as an unsigned long specifying the number of bytes. This option is read-only.

TCP_KEEPALIVE Activates the keep-alive timer. (UInt32). The default setting is off.

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;
    long    kp_timeout;     /
};
```

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        /
#define UDP_RX_ICMP             0x2
```

Option descriptions

UDP_CHECKSUM Activate or deactivate a checksum calculation. (UInt32)

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
#define IP_TOS                      0x02
#define IP_TTL                      0x03
#define IP_REUSEADDR                0x04
```

```
#define IP_DONTROUTE                0x10
#define IP_BROADCAST                0x20
#define IP_HDRINCL                  0x1002
#define IP_RCVOPTS                  0x1005
#define IP_RCVDSTADDR               0x1007
#define IP_MULTICAST_IF             0x1010
#define IP_MULTICAST_TTL            0x1011
#define IP_MULTICAST_LOOP           0x1012
#define IP_ADD_MEMBERSHIP           0x1013
#define IP_DROP_MEMBERSHIP          0x1014  /
#define IP_BROADCAST_IF             0x1015
#define IP_RCVIFADDR                0x1016
```

Constants for use with the IP_TOS precedence level are as follows:

```
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
};
```

Constants for use with the IP_TOS type of service are as follows:

```
/*  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. (UInt8[])

This option is intended for use by network debugging and control programs; most applications do not need this option.

The value for this option consists of a string of bytes 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 routers (as returned by `OTInetGetInterfaceInfo`) must be for the first-hop router. Open Transport extracts the first-hop router address from the option list and adjusts the size of the list before transmitting the packet. The Options field can contain up to 40 bytes.

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 this option for the received IP datagram. The `OTRcvUDErr` function returns this option for 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. (`UInt8`)

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 `SET_TOS` macro (page 685) to construct this option. 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 this option for 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. (`UInt8`)

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 way to get the Time to Live field of an incoming datagram.

`IP_REUSEADDR` Allow multiple addresses with the same port number. (`UInt32`)

Set this option to `T_YES` to allow TCP to bind a transport endpoint to a `kOTAnyFreeAddress` 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. (`UInt32`)

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. (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. (`UInt32`)

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. (`UInt32`)

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. See "Using RawIP" (page 247) for more information.

IP_RCVOPTS
Include IP-level options when you call the OTRcvUData function. (UInt32)

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. (UInt32)

If you specify T_YES for this option, the OTRcvUData function includes the address of the interface. If you specify T_NO (the default) for this option, 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. (UInt32)

Specify the interface as an "InetHost". 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 or the number of hops permitted for outgoing multicast IP datagrams, or retrieve the Time to Live field set for an interface. (UInt8)

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. (UInt32)

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. (SizeOf(TIPAddMulticast)).

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" (page 690).

IP_DROP_MEMBERSHIP

Drop membership in an IP multicast group.

(SizeOf(TIPAddMulticast)).

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 and the TCP/IP utility functions. In addition to these functions, you need the functions described in the chapter "Endpoints" (page 683), in order to implement TCP/IP communications.

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. The name should be a null-terminated string.
`hinfo`	A pointer to an `InetHostInfo` structure (page 689) 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.
function result	A result code. See Appendix B for more information.

DISCUSSION

A single host can be associated with multiple internet addresses. You can use the OTInetStringToAddress function to return multiple addresses for multihomed hosts.

If you specify an internet address in dotted-decimal format for the name 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.

The OTLookupName function (page 559) 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 686) 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”(page 421). SPECIAL CONSIDERATIONS

If you call the OTInetStringToAddress function asynchronously, do not modify the InetHostInfo structure until the function completes.

SEE ALSO

The OTInetAddressToName function (page 701).

The OTInetHostToString function (page 715).

OTInetAddressToName

Determines the canonical name of the host associated with an IP 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 .
`name`	A character array that you must allocate. On output, the function places the canonical name in this array.
function result	A result code. See Appendix B for more information.

DISCUSSION

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 the `name` parameter to this call. 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.

SEE ALSO

The `OTInetStringToAddress` function (page 700).

The `OTInetStringToHost` function (page 715).

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`.

Use the `OTCloseProvider` function, described in the chapter "Endpoints" (page 683), to close a TCP/IP service provider when you are finished using it.

OTAsyncOpenInternetServices

Opens the TCP/IP service provider and returns an Internet services reference.

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. You can obtain the default configuration 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.

function result A result code. See Appendix B for additional information.

DISCUSSION

You must open a TCP/IP service provider before calling any TCP/IP services functions other than the address utility functions. You must provide the Internet services reference when calling any of these non-utility functions. The `OTAsyncOpenInternetServices` function runs asynchronously and sets the mode of the TCP/IP service provider as asynchronous.

For information on how to specify options using the `cfig` parameter, see Table 11-2 (page 246).

SEE ALSO

The `OTOpenInternetServices` function (page 704).

OTOpenInternetServices

Opens the TCP/IP service provider and returns an internet services reference.

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. You can obtain the default configuration by using the constant `kDefaultInternetServicesPath` for this parameter.

`oflag`	Reserved. Must be set to 0.
`err`	A pointer to the function result.
function result	An internet services reference.

DISCUSSION

You must open a TCP/IP service provider before calling any TCP/IP service functions 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 runs synchronously and also sets the mode of the TCP/IP service provider 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 11-2 (page 246) provides the constant name used in "Options" (page 691) and the value to used in the configuration string

SEE ALSO

The `OTAsyncOpenInternetServices` function (page 703).

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(page 690). You must allocate this structure. The function fills it in with the processor type and operating-system version of the host.

function result A result code. See Appendix B for more information.

DISCUSSION

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 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 modify the `InetSysInfo` structure until the function completes. The information is held in the domain name server's database and it is often incomplete and inaccurate.

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(page 691). You must allocate the structures in this array.
function result	A result code. See Appendix B for more information.

DISCUSSION

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 mail gateway. 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.

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 modify the `InetMailExchange` array until the function completes.

SEE ALSO

The `InetMailExchange` structure (page 691).

Internet mail routing and mail-exchange resource records are described in RFC 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

Executes a generic DNS query.

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 issuing the query.

`qClass` The numeric value for the DNS resource record class, such as `Inet` or `Hesiod`. 255 is a wildcard.

`qType` The numeric value for the DNS resource record type, such as `CNAME` and `PTR`,. 255 is a wildcard.

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 uses to return a set of pointers to the individual DNS query information structures returned. This parameter is optional, specify nil if you don't want to use this array.
argvlen	The length of the argv buffer.
flags	Reserved. Set to 0.
function result	A result code. See Appendix B for more information.

DISCUSSION

The OTInetQuery function allows you to use the Domain Name Resolver (DNR) for generic domain name service (DNS) queries. You can ask query any 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 DNSQueryInfo structures written into the buffer pointed to by the buf parameter. For example, if you set argvlen to 5 and your query receives three answers, 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. When using asynchronous mode, you must not modify 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.

Open Transport provides simplified functions 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(page 700)`, `OTInetAddressToName(page 701)`, `OTInetSysInfo(page 705)`, and `OTInetMailExchange` (page 707)functions, respectively. 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` are cached by OpenTransport. OpenTransport does not use this cached information to resolve address-to-name translations because doing so would defeat server load balancing schemes.

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 internet interfaces on 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 (page 688). You must allocate this structure. The function fills in the structure for the internet interface indicated by the `val` 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.
function result	A result code. See Appendix B for more information.

DISCUSSION

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 router (that is, a router , 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.

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 deselected “load only when needed” in the TCP/IP control panel.

The `OTInetGetInterfaceInfo` function always completes immediately.

Open Transport does not currently support more than one IP interface.

SEE ALSO

The `OTBind` function (page 441).

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 (page 685) that you allocate. The function fills in this structure.

port — The port number of the address.

host — The IP address of the host.

DISCUSSION

This function copies the host parameter to the fHost field of the InetAddress, and the port parameter to the fPort field.

This function also 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(page 484), OTSndURequest(page 469), and OTBind(page 441). You are not required to use the OTInitInetAddress function when creating an InetAddress structure; this function is provided for your convenience only.

SPECIAL CONSIDERATIONS

The OTInetInitInetAddress function cannot block and always runs synchronously. It does not use the services of the DNR.

SEE ALSO

The OTInetStringToHost function (page 715).

OTInitDNSAddress

Fills in a DNSAddress structure with your data.

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 (page 686)that you allocate. The function fills in the fAddressType field of the DNSAddress structure with the value AF_DNS and fills in the structure's fName field with the address string you specify.
str	A pointer to a domain name string.
function result	The function returns the size of the resulting DNSAddress structure as an unsigned integer.

DISCUSSION

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.

SPECIAL CONSIDERATIONS

The OTInetDNSAddress function cannot block and always runs synchronously. It does not use the services of the DNR.

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 (page 683).
function result	A result code. See Appendix B for more information.

SPECIAL CONSIDERATIONS

The OTInetStringToHost function cannot block and always runs synchronously. It does not use the services of the DNR.

SEE ALSO

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`	The address as an `InetHost` data type(page 689).
`str`	A pointer to a C stringthat , on return, contains 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 `OTInetStringToHost` function (page 715).

Single Link Multi-Homing

You can use the `OTInetGetSecondaryAddress` function to determine the secondary IP addresses supported for your machine.

OTInetGetSecondaryAddress

Returns the active secondary IP addresses.

C INTERFACE

```
void OTInetGetSecondaryAddress (InetHost* addr,
                                UInt32* count,
                                SInt32 index);
```

C++ INTERFACE

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

PARAMETERS

`addr` A pointer to a buffer into which the secondary IP addresses are placed when the function returns.

`count` The number of secondary IP address that must fit in the buffer referenced by the `address` parameter. You obtain this number by examining the `fIPSecondaryCount` field of the `InetInterfaceInfo` structure (page 688).

`index` Specifies the IP interface for which to obtain secondary addresses. For the primary IP interface,the index is 0.

DISCUSSION

This function copies the supported secondary addresses associated with an IP interface. The count parameter specifies how many secondary address to return in the buffer referenced by the addr parameter. This buffer must be of the size `count * sizeof(InetAddr)` to return all of the available addresses. Use the value stored in the `fIPSecondaryCount` field of the `InetInterfaceInfo` structure for the `count` parameter. The `OTInetGetSecondaryAddress` function modifies the `count` value to indicate that the number of secondary addresses actually returned is less than the specified number of secondary addresses.

AppleTalk Reference

Contents

This chapter provides reference information for AppleTalk providers and for options used with AppleTalk endpoints.

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 use the constants in this section to define the length of AppleTalk addresses and NBP name strings and to identify wildcards used in NBP names.

The constant `kNBPEntityBufferSize` specifies the maximum size of the NBP name buffer, currently defined to be 105 bytes. This permits an 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” (page 723) and “The Combined DDP-NBP Address Structure” (page 724) 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           = sizeof(DDPAddress),
    kNBPAddressLength           = kNBPEntityBufferSize,
    kAppleTalkAddressLength     = kDDPAddressLength + kNBPEntityBufferSize
};
```

Address Format Constants

You use the following constants to identify each AppleTalk address structure.

```
enum {
    AF_ATALK_DDP            = 0x0100,
    AF_ATALK_DDPNBP         = 0x0101,
    AF_ATALK_NBP            = 0x0102,
    AF_ATALK_MNODE          = 0x0103
};
```

Field descriptions

`AF_ATALK_DDP` DDP address type.

`AF_ATALK_DDPNBP` DDPNBP address type.

`AF_ATALK_NBP` NBP address type.

`AF_ATALK_NNODE` Multinode address type.

The DDP Address Structure

You use the DDP address format, specified by the DDP address structure, to identify the 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;
};
typedef struct DDPAddress DDPAddress;
```

FIELD DESCRIPTIONS

`fAddressType`
: A number that specifies the format of the address. For a DDP address this is always 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 the broadcast address; 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. 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)" (page 303)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];
};
typedef struct NBPAddress NBPAddress;
```

FIELD DESCRIPTIONS

`fAddressType`

A number that specifies the format of the address. For an NBP address this is the constant `AF_ATALK_NBP`.

`fNBPNameBuffer`

An array of chars 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

When you bind an endpoint you use the combined DDP-NBP address format, specified by the combined DDP-NBP address structure, to identify the socket address and the NBP name associated with your endpoint. The combined DDP-NBP address structure is defined by the `DDPNBPAddress` data type.

```
struct DDPNBPAddress
{
    OTAddressType       fAddressType;
    UInt16              fNetwork;
    UInt8               fNodeID;
    UInt8               fSocket;
    UInt8               fDDPType;
    UInt8               fPad;
    UInt8               fNBPNameBuffer[kNBPEntityBufferSize];
};
typedef struct DDPNBPAddress DDPNBPAddress;
```

FIELD DESCRIPTIONS

`fAddressType` A number that specifies the format of the address. For a DDPNBPAddress, this is always 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 the broadcast address; 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. 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)"(page 303), and *Inside AppleTalk,* second edition.

`fPad` Reserved. Set to 0.

`fNBPNameBuffer`
An array of chars 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, which also uses the DDP address structure, to identify the 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" (page 722).

```
struct DDPAddress
{
    OTAddressType       fAddressType;
    UInt16              fNetwork;
    UInt8               fNodeID;
    UInt8               fSocket;
    UInt8               fDDPType;
    UInt8               fPad;
};
typedef struct DDPAddress DDPAddress;
```

FIELD DESCRIPTIONS

fAddressType A number that specifies the format of the address. For a multinode address, this is always 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 the broadcast address; 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` Ignored for multinode addresses.

`fDDPType` Ignored for multinode addresses.

`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];
};
typdef struct NBPEntity NBPEntity;
```

FIELD DESCRIPTIONS

`fEntity` An array of chars that specifies the NBP entity you wish to use to hold the NBP name.

The NBP entity holds an NBP name as three packed Pascal strings, three closely packed Pascal strings. 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 quoted escape characters, but the NBP entity extraction functions add them as necessary when converting NBP name strings from NBP entities.

AppleTalk Utility Functions

This section describes AppleTalk utility functions that initialize DDP and NBP data structures, compare DDP addresses, and transfer data between NBP entities and NBP names.

OTInitDDPAddress

Initializes a DDP address structure.

C INTERFACE

```
size_t OTInitDDPAddress(DDPAddress* address,
                    UInt16 net,
                    UInt8 node,
                    UInt8 socket,
                    UInt8 ddpType);
```

C++ INTERFACE

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

PARAMETERS

`address`	A pointer to the DDP address structure (page 722) 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.

function result The size of the NBP address structure.

SEE ALSO

The `OTInitNBPAddress` function (page 729).

The `OTInitDDPNBPAddress` function (page 730).

"Datagram Delivery Protocol" (page 303).

OTInitNBPAddress

Initializes an NBP address structure.

C INTERFACE

```
size_t OTInitNBPAddress(NBPAddress* address,
                        const char* name);
```

C++ INTERFACE

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

PARAMETERS

`address`	A pointer to the NBP address structure (page 723) you wish to initialize.
`name`	A pointer to the NBP string you wish to use for the NBP name.
function result	The size of the NBP address structure.

DISCUSSION

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—that is, *name:type@zone*, with special characters quoted.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 `OTInitDDPAddress` function (page 728).

The `OTInitDDPNBPAddress` function (page 730).

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++ INTERFACE

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

PARAMETERS

`address`	A pointer to the combined DDP-NBP address structure (page 724) 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.

function result The size of the combined DDP-NBP address structure.

DISCUSSION

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 `OTInitNBPAddress` function (page 729).

The `OTInitDDPAddress` function (page 728).

"Datagram Delivery Protocol (DDP)" (page 303).

OTCompareDDPAddresses

Compares two DDP address structures.

C INTERFACE

```
Boolean OTCompareDDPAddresses(const DDPAddress* addr1,
                              const DDPAddress* addr2);
```

C++ INTERFACE

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.
function result	A value of `true` means the two addresses match.

DISCUSSION

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 (except for the node field, which must match exactly).

OTInitNBPEntity

Initializes an NBP entity structure.

C INTERFACE

```
void OTInitNBPEntity(NBPEntity* nbpEntity);
```

C++ INTERFACE

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

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure (page 727) you wish to initialize.

DISCUSSION

The OTInitNBPEntity function initializes an NBP entity structure, setting the name, type and zone parts of an NBP name to empty strings.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the OTSetNBPType function (page 739), the OTSetNBPType function (page 739), and the OTSetNBPZone function (page 740), respectively.

OTGetNBPEntityLengthAsAddress

Obtains the size an NBP entity structure would be if it were formatted as a string.

C INTERFACE

```
size_t OTGetNBPEntityLengthAsAddress(const NBPEntity* nbpEntity);
```

C++ INTERFACE

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

PARAMETERS

nbpEntity A pointer to the NBP entity structure (page 727) you wish to determine the length of.

function result The size of an NBP entity structure.

DISCUSSION

The OTGetNBPEntityLengthAsAddress function obtains the number of bytes needed to store an NBP entity structure into the name part of an NBP or combined DDP-NBP address structure.

To store an NBP entity structure as an NBP address string, use the OTSetAddressFromNBPEntity function (page 734).

SPECIAL CONSIDERATIONS

Use this function to determine the appropriate buffer size for an NBP entity before using the OTSetAddressFromNBPEntity function.

SEE ALSO

"The NBP Address Structure" (page 723).

"The Combined DDP-NBP Address Structure" (page 724).

OTSetAddressFromNBPEntity

Stores an NBP entity structure as an NBP address string.

C INTERFACE

```
size_t OTSetAddressFromNBPEntity(UInt8* nameBuf,
                                 const NBPEntity* entity);
```

C++ INTERFACE

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

PARAMETERS

nameBuf	A pointer to a stroing in which you wish to store the NBP entity, formatted as a string.
entity	A pointer to the NBP entity (page 727) you wish to store.
function result	The number of bytes that were actually stored in the buffer referenced by the nameBuf parameter.

DISCUSSION

The OTSetAddressFromNBPEntity function stores the information in the NBP entity into the string 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.

To determine the appropriate buffer size for this call, use the `OTGetNBPEntityLengthAsAddress` function (page 733).

To parse and store all or part of an NBP name into an NBP entity, use the `OTSetNBPEntityFromAddress` function (page 735).

SPECIAL CONSIDERATIONS

Use the `OTGetNBPEntityLengthAsAddress` function beforehand to determine the appropriate buffer size.

SEE ALSO

"The NBP Address Structure" (page 723).

OTSetNBPEntityFromAddress

Parses and stores an NBP address ("NBP Address" or "DDPNBPAddress") into an NBP entity.

C INTERFACE

```
Boolean OTSetNBPEntityFromAddress(NBPEntity* entity,
                                  const UInt8* addrBuf,
                                  size_t len);
```

C++ INTERFACE

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

PARAMETERS

`nbpEntity` A pointer to the NBP entity (page 727) in which you wish to store the address.

`addrBuf` A pointer to the address buffer from which to get the NBP name string.

`len` The number of characters in the address buffer

function result A value of `true` means the function succeeded. See Discussion.

DISCUSSION

The `OTSetNBPEntityFromAddress` function parses the string part of 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 later be 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, the function still stores the data, but in a truncated form.

To copy the contents of an NBP entity into an NBP address structure, use the `OTSetAddressFromNBPEntity` function (page 734).

To determine the appropriate buffer size for an NBP entity, use the `OTGetNBPEntityLengthAsAddress` function (page 733).

To store the NBP name in an NBP entity, use the `OTSetNBPName` function (page 738); to store the NBP type, use the `OTSetNBPType` function (page 739); and to store the NBP zone, use the `OTSetNBPZone` function (page 740).

To extract the name portion of an NBP name from an NBP entity, use the `OTExtractNBPName` function (page 741); to extract the type portion of an NBP name, use the `OTExtractNBPType` function (page 742); and to extract the zone portion, use the `OTExtractNBPZone` function (page 744).

SEE ALSO

"The NBP Address Structure" (page 723).

"The Combined DDP-NBP Address Structure" (page 724).

OTSetAddressFromNBPString

Copies an NBP name string into an NBP address buffer.

C INTERFACE

```
size_t OTSetAddressFromNBPString(UInt8* addrBuf,
                                 const char* nbpName,
                                 SInt32 len);
```

C++ INTERFACE

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. Specify -1 to copy all the characters in the string up to a null terminator. Use -1 to copy the entire string.
function result	The number of bytes actually copied.

DISCUSSION

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.

To copy the contents of an NBP entity into an NBP address structure, use the `OTSetAddressFromNBPEntity` function (page 734).

The `nbpName` parameter must have a "\" in front of all the special characters.

SEE ALSO

"The NBP Address Structure" (page 723).

OTSetNBPName

Set the name part of an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPName(NBPEntity* entity,
                     const char* name);
```

C++ INTERFACE

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

PARAMETERS

`nbpEntity`	A pointer to the NBP entity structure (page 727) in which you wish to set the name .
`name`	A pointer to the name portion of an NBP name string that you wish to store. Special characters must be quoted.
function result	See Discussion.

DISCUSSION

The `OTSetNBPName` function stores the NBP name specified by the `name` parameter into the NBP entity structure indicated by the `entity` 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).

To store the type and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPType` function (page 739) and the `OTSetNBPZone` function (page 740), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 741), the `OTExtractNBPType` function (page 742), and the `OTExtractNBPZone` function (page 744), respectively.

SPECIAL CONSIDERATIONS

Backslashes in front of special characters will be removed. The function `OTSetAddressFromNBPEntity` will automatically insert them for mapper calls.

SEE ALSO

"The NBP Address Structure" (page 723).

OTSetNBPType

Set the type part of an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPType(NBPEntity* entity,
                     const char* typeVal);
```

C++ INTERFACE

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

PARAMETERS

`entity` A pointer to the NBP entity structure (page 727) in which you wish to set the type.

`type` A pointer to the type portion of an NBP name string that you wish to store. Special characters must be quoted.

function result The function returns `false` if the `type` parameter is longer than the maximum allowed for type part of an NBP name (32 characters).

DISCUSSION

The OTSetNBPType function stores the NBP type specified by the typeVal parameter into the NBP entity structure indicated by the entity parameter, deleting any previous type stored there. To store the name and zone parts of an NBP name in an NBP entity structure, use the OTSetNBPName function (page 738) and the OTSetNBPZone function (page 740), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the OTExtractNBPName function (page 741), the OTExtractNBPType function (page 742), and the OTExtractNBPZone function (page 744), respectively.

SPECIAL CONSIDERATIONS

Backslashes in front of special characters will be removed. The function OTSetAddressFromNBPEntity will automatically insert them for mapper calls.

SEE ALSO

"The NBP Address Structure" (page 723).

OTSetNBPZone

Set the zone part of an NBP entity structure.

C INTERFACE

```
Boolean OTSetNBPZone(NBPEntity* entity,
                     const char* zone);
```

C++ INTERFACE

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

PARAMETERS

entity	A pointer to the NBP entity structure (page 727) in which you wish to set the zone.

`zone` A pointer to the zone portion of an NBP name string that you wish to store. Special characters must be quoted.

function result This function returns `false` if the `zone` parameter is longer than the maximum allowed for zone part of an NBP name (32 characters).

DISCUSSION

The `OTSetNBPZone` function stores the NBP zone specified by the `zone` parameter into the NBP entity structure indicated by the `entity` parameter, deleting any previous zone stored there. To store the name and type parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 738) and the `OTSetNBPType` function (page 739), respectively.

To extract the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 741), the `OTExtractNBPType` function (page 742), and the `OTExtractNBPZone` function (page 744), respectively.

SPECIAL CONSIDERATIONS

Backslashes in front of special characters will be removed. The function `OTSetAddressFromNBPEntity` will automatically insert them for mapper calls.

SEE ALSO

“The NBP Address Structure” (page 723).

OTExtractNBPName

Extracts the name part of an NBP name from an NBP entity structure.

C INTERFACE

```
void OTExtractNBPName(const NBPEntity* entity,
                      char* name);
```

C++ INTERFACE

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

PARAMETERS

`entity`	A pointer to the NBP entity structure (page 727) from which you wish to extract the name.
`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.

DISCUSSION

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.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 738), the `OTSetNBPType` function (page 739), and the `OTSetNBPZone` function (page 740), respectively.

To extract the type and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPType` function (page 742) and the `OTExtractNBPZone` function (page 744), respectively.

SEE ALSO

“The NBP Address Structure” (page 723).

OTExtractNBPType

Extracts the type part of an NBP entity structure.

C INTERFACE

```
void OTExtractNBPType(const NBPEntity* entity,
                      char* typeVal);
```

C++ INTERFACE

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

PARAMETERS

`entity` A pointer to the NBP entity structure (page 727) from which you wish to extract the type.

`typeVal` 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.

DISCUSSION

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.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 738), the `OTSetNBPType` function (page 739), and the `OTSetNBPZone` function (page 740), respectively.

To extract the name and zone parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 741) and the `OTExtractNBPZone` function (page 744), respectively.

Note
If the `name` parameter passed to the function `OTSetNBPName` contains a backslash, the result in the `name` parameter passed to the function `OTExtractNBPName` will not be the same, because the backslash is stripped by the function `OTSetNBPName` and is not re-added by the function `OTExtractNBPName`. ◆

SEE ALSO

“The NBP Address Structure” (page 723).

OTExtractNBPZone

Extracts the zone part of an NBP name from an NBP entity structure.

C INTERFACE

```
void OTExtractNBPZone(const NBPEntity* entity,
                      char* zone);
```

C++ INTERFACE

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

PARAMETERS

`entity` A pointer to the NBP entity structure (page 727) from which you wish to extract the zone.

`zone` A pointer to the string buffer in which to store the zone portion of an NBP name string that you wish to extract from the NBP entity.

DISCUSSION

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.

To store the name, type, and zone parts of an NBP name in an NBP entity structure, use the `OTSetNBPName` function (page 738), the `OTSetNBPType` function (page 739), and the `OTSetNBPZone` function (page 740), respectively.

To extract the name and type parts of an NBP name in an NBP entity structure, use the `OTExtractNBPName` function (page 741) and the `OTExtractNBPType` function (page 742) , respectively.

Note
If the name parameter passed to the function OTSetNBPName contains a backslash, the result in the name parameter passed to the function OTExtractNBPName will not be the same, because the backslash is stripped by the function OTSetNBPName and is not re-added by the function OTExtractNBPName. ◆

SEE ALSO

"The NBP Address Structure" (page 723).

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 all Open Transport 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 with a completion event. The prototype for the notifier function is the same as for the notifier you use for your endpoint providers.

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 the name of the zone.

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 755) to obtain the AppleTalkInfo data.

AppleTalk Service Functions

You use the AppleTalk service provider functions to open an AppleTalk service provider, and to obtain information about zones and about the network to which your node is connected.

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.
function result	See Discussion.

DISCUSSION

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 the AppleTalk service provider 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` operation completes, Open Transport calls the notifier function identified in the `proc` parameter. Open Transport passes an AppleTalk service provider reference in your notifier's cookie parameter and passes 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.

To close the AppleTalk service provider, call the OTCloseProvider function, described in the chapter "Providers Reference" in this book.

SPECIAL CONSIDERATIONS

When you no longer need to use AppleTalk service provider functions, you must call the generic Open Transport function OTCloseProvider(page 392).

SEE ALSO

The OTOpenAppleTalkServices function (page 749).

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 (`kOTNoErr`) indicates successful completion.
function result	See Discussion.

DISCUSSION

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 the AppleTalk service provider to synchronous.

SPECIAL CONSIDERATIONS

When you no longer need to use AppleTalk service provider functions, you must call the generic Open Transport function `OTCloseProvider`(page 392).

SEE ALSO

The `OTAsyncOpenAppleTalkServices` function (page 747).

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.

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 `TNetbuf` structure that, on return, contains your application's AppleTalk local zone name.
function result	A result code. See Appendix B for more information.

DISCUSSION

The `OTATalkGetMyZone` function returns the name of your application's AppleTalk zone (as a Pascal string) in the buffer referenced by the `zone` parameter. 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.

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.

Note
Using a Pascal string for the zone name is redundant since you can determine the length of the string from the `len` field of the `TNetbuf` structure, but the other zone-related calls use Pascal strings, so this call also uses them for consistency. ◆

SEE ALSO

The `OTATalkGetLocalZones` function (page 752).

The `OTATalkGetZoneList` function (page 753).

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 `TNetbuf` structure that, on return, contains a list of the local zone names.
function result	See Discussion.

DISCUSSION

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, stored as packed Pascal strings.

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 are packed. 6 KB bytes is likely to be a safe value for the buffer's size.

SEE ALSO

The OTATalkGetMyZone function (page 751).

The OTATalkGetZoneList function (page 753).

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.

function result A result code. See Appendix B for information.

DISCUSSION

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 string.

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 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.

SEE ALSO

The `OTATalkGetMyZone` function (page 751).

The `OTATalkGetLocalZones` function (page 752).

Obtaining Information About Your AppleTalk Environment

The `OTATalkGetInfo` function provides information about the current AppleTalk environment for your 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);
```

PARAMETERS

`ref`	The reference value of your AppleTalk service provider.
`info`	A pointer to a `TNetbuf` structure that, on return, contains information about your current AppleTalk environment. You should allocate the buffer referenced by this structure using the `AppleTalkInfo` type, and you should set the buffer size using `sizeof(AppleTalkInfo)`.
function result	A result code. See Appendix B for more information.

DISCUSSION

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 the `info` parameter, which points in turn to a buffer containing the `AppleTalkInfo` structure. The maximum size of this buffer is the size of the AppleTalk information structure.

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.

The AppleTalkInfo data structure is described in the section "Constants and Data Types" (page 745).

DDP Reference

This section describes the options that are specific to DDP, defines the constant you use to specify the DDP protocol for option management functions, and indicates the generic Open Transport options you can use with DDP.

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(page 572) 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
```

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 OTOptionManagement function(page 575). 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.

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.

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 (page 572) 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 End-of-Message Option" (page 317) and in "The Checksum Option" (page 318).

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
#define ATP_OPT_DATALEN     0x2111
#define ATP_OPT_RELTIMER    0x2112
#define ATP_OPT_TRANID      0x2113
```

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 if it is different from the ATP default of 578; only the PAP server uses this option. The ATP_OPT_TRANID option sets 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 (page 572) 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 “Specifying ATP Options” (page 329).

PAP Reference

This section describes the 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

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, used with the OTConnect provider function (page 484), forces PAP to retry opening a connection a specific number of times before failing.

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 also use the generic OPT_ENABLEEOM, OPT_CHECKSUM and OPT_SERVERSTATUS options, which are described in “Specifying PAP Options” (page 339).

CHAPTER 31

Serial Endpoint Reference

Contents

This chapter 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 (page 376) to configure a serial endpoint.

#define `kSerialName`'serial'
#define `kSerialPortAName`'serialA'
#define `kSerialPortBName`'serialB'
#define `kSerialPortABName`'serialAB'

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 (page 592), described in the chapter "Ports Reference" 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 the constants listed below.

Use these constants with `I_SetSerialDTR`:

```
kOTSerialSetDTROn               = 1             /* Turn the DTR signal on */
kOTSerialSetDTROff              = 0             /* Turn the DTR signal off */
```

```
Use these constants with I_SetSerialBreak:

kOTSerialSetBreakOn     = 0xffffff     /* Turn the break signal on */
kOTSerialSetBreakOff    = 0            /* Turn the break signal off */
```

Use these constants with `I_SetSerial/XOFFState`:

```
kOTSerialForceXOffTrue      = 1     /* Unconditional set XOFF state */
kOTSerialForceXOffFalse     = 0     /* Unconditional clear XOFF state */
```

Use these constants with I_SetSerial/XONState:

```
kOTSerialSendXOnAlways      = 1     /* Always send XON character */
kOTSerialSendXOnIfXOffTrue  = 0     /* Send XON char if XOFF state */
```

Use these constants with I_SetSerial/XOFFState:

```
kOTSerialSendXOffAlways     = 1         /* Always send XOFF character */
kOTSerialSendXOffIfXOnTrue      = 0     /* Send XOFF char 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  = 1024,            /* send buffer = 1024characters */
    kOTSerialDefaultRcvBufSize  = 1024,             /* recv buffer = 1024characters */
    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" (page 575).

```
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.

Parity	Value	Description
kOTSerialNoParity	0	No parity. (default)
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.

Status	Value	Description
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. Use the SerialHandShakeData macro to construct this value.

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—that is, error characters are discarded. 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 value of 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. 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 17 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);
```

Note

Note that the on value is 0 and the off value is 1. ◆

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.

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 such as 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.

You must use 4 byte ints in your development environment. ▲

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`

Table A-1 XTI-to-Open Transport function cross-reference (continued)

XTI function	Open Transport function
t_open	OTOpenEndpoint
t_optmgmt	OTOptionManagement
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 shows the correspondence between Open Transport functions and XTI functions. If the Open Transport function is not in the list, it is not suppoerted by the XTI interface standard. It might be supported by an Apple extension to XTI, listed in Table A-3.

Table A-2 Open Transport-to-XTI function cross-reference

Open Transport function	XTI function
OTAccept	t_accept
OTAlloc	t_alloc
OTBind	t_bind

Table A-2 Open Transport-to-XTI function cross-reference (continued)

Open Transport function	XTI function
OTCloseProvider	t_close
OTConnect	t_connect
DontAckSends	—
OTFree	t_free
OTGetEndpointInfo	t_getinfo
OTGetProtAddress	t_getprotaddr
OTGetEndpointState	t_getstate
OTListen	t_listen
OTLook	t_look
OTOpenEndpoint	t_open
OTOptionManagement	t_optmgmt
OTRcv	t_rcv
OTRcvConnect	t_rcvconnect
OTRcvDisconnect	t_rcvdis
OTRcvOrderlyDisconnect	t_rcvrel
OTRcvUData	t_rcvudata
OTRcvUDErr	t_rcvuderr
OTSnd	t_snd
OTSndDisconnect	t_snddis
OTSndOrderlyDisconnect	t_sndrel
OTSndUData	t_sndudata
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. If a function is not listed in this table or in Table A-2, it is not available to XTI clients.

Table A-3 Open Transport Functions not found in XTI

Open Transport preferred-C name	XTI-style name
OTCancelRequest	t_cancelrequest
OTCancelReply	t_cancelreply
OTCancelSynchronousCalls	t_cancelsynchronouscalls
OTCancelURequest	t_cancelurequest
OTCancelUReply	t_cancelureply
OTGetProtAddress	t_getprotaddr
OTInstallNotifier	t_installnotifier
OTIsNonBlocking	t_isnonblocking
OTIsSynchronous	t_issynchronous
OTRcvRequest	t_rcvrequest
OTRcvReply	t_rcvreply
OTRcvUReply	t_rcvureply
OTRcvURequest	t_rcvurequest
OTRemoveNotifier	t_removenotifier
OTResolveAddress	t_resolveaddr
OTSetAsynchronous	t_asynchronous
OTSetBlocking	t_blocking

Table A-3 Open Transport Functions not found in XTI (continued)

Open Transport preferred-C name	XTI-style name
OTSetNonBlocking	t_nonblocking
OTSetSynchronous	t_synchronous
OTSndReply	t_sndreply
OTSndRequest	t_sndrequest
OTSndUReply	t_sndureply
OTSndURequest	t_sndurequest
OTUseSyncIdleEvents	t_usesynchidleevents

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

Table A-4 XTI-to-Open Transport data structure cross-reference (continued)

XTI name	Open Transport name
t_unitdata	TUnitData
t_uderr	TUDErr
t_optmgmt	TOptMgmt

Table A-5 lists the apple extensions to XTI data structures.

Table A-5 Apple extensions to XTI data structures

Open Transport name	XTI name
TOptionhdr	t_opthdr
TRequest	t_request
TReply	t_reply
TUnitRequest	t_unitrequest
TUnitReply	t_unitreply

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-6. For an explanation of Open Transport result codes, see Appendix B (page 785).

Table A-6 XTI-to-Open Transport result code cross-reference

XTI result code	Open Transport result code
TACCES	kOTAccessErr
TADDRBUSY	kOTAddressBusyErr
TBADADDR	kOTBadAddressErr
TBADDATA	kOTBadDataErr
TBADF	kOTBadReferenceErr
TBADFLAG	kOTBadFlagErr
TBADNAME	kOTBadNameErr
TBADOPT	kOTBadOptionErr
TBADQLEN	kOTBadQLenErr
TBADSEQ	kOTBadSequenceErr
TBADSYNC	kOTBadSyncErr
TBUFOVFLW	kOTBufferOverflowErr
TCANCELED	kOTCanceledErr
TFLOW	kOTFlowErr
TINDOUT	kOTIndOutErr
TLOOK	kOTLookErr
TNOADDR	kOTNoAddressErr
TNODATA	kOTNoDataErr
TNODIS	kOTNoDisconnectErr
TNOREL	kOTNoReleaseErr
TNOSTRUCTYPE	kOTStructureTypeErr
TNOTSUPPORT	kOTNotSupportedErr
TNOUDERR	kOTNoUDErrErr

Table A-6 XTI-to-Open Transport result code cross-reference (continued)

XTI result code	Open Transport result code
TOUTSTATE	kOTOutStateErr
TPROTO	kOTProtocolErr
TPROVMISMATCH	kOTProviderMismatchErr
TQFULL	kOTQFullErr
TRESADDR	kOTResAddressErr
TRESQLEN	kOTResQLenErr
TSTATECHNG	kOTStateChangeErr
TSUCCESS	kOTNoErr
TSYSERR	kOTSysErrorErr

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	0	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 provider reference does not refer to a valid provider.
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.

Table B-1 Open Transport result codes (continued)

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; your notifier function will also get an asynchronous event. 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 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 you try to read data but the endpoint is in asynchronous, or is in nonblocking mode, and no data is currently available. For a mapper, this result is returned by the OTLookupName function when no names are found.
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.

Table B-1 Open Transport result codes (continued)

Result code	Value	Meaning
kOTNotSupportedErr	–3167	This action is not supported by this endpoint.
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 XTI 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 or OTFree 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.

Table B-1 Open Transport result codes (continued)

Result code	Value	Meaning
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 connection-oriented endpoints to the same address and with a qlen field greater than 0.
kOTIndOutErr	–3173	There are outstanding connection indications on the endpoint, and you are accepting a connection on this endpoint. When accepting a connection on 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.
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.

Table B-1 Open Transport result codes (continued)

Result code	Value	Meaning
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. This is usually fatal. Normal recovery is to close the provider.
kOTBadSyncErr	–3179	You attempted a synchronous call at hardware or deferred task level, or you made an Open Transport call at hardware interrupt time. Note that Open Transport cannot always detect this condition, so you cannot rely on getting this error. If undetected, your system could crash.
kOTCanceledErr	–3180	A provider function never finished executing because the provider was closed or because the function was synchronous and synchronous functions were cancelled.
kOTNotFoundErr	–3201	Requested information does not exist.
kENOENTErr		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, or to operate on any other entity that was not found.
kENIOErr	–3204	An I/O error occurred.
kENXIOErr	–3205	No such device or address.
kEBADFErr	–3208	The provider reference or stream reference supplied to the function was not valid.
kEAGAINErr	–3210	A provider is in non-blocking mode and cannot perform this operation now; Open Transport would have to block to complete the request. Try again later.

Result code	Value	Meaning
kENOMEMErr	–3211	Open Transport cannot allocate enough memory to meet your request.
kOTOutOfMemoryErr		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	You attempted an invalid operation, or you passed an invalid parameter.
kEWOULDBLOCKErr kEDEADLKErr	–3234	In order to complete the requested operation, the endpoint provider would have to block, and the endpoint is in nonblocking mode.
kEADDRINUSEErr	–3247	The address is in use and is not available for the current function.
kEADDRNOTAVAILErr	–3248	The address is not available or the requested address is not appropriate for the current function because the function requires a multicast address.
kENETDOWNErr	–3249	TCP/IP error. The path to a network number is currently unavailable.
kENETUNREACHErr	–3250	TCP/IP error. The path to a network number does not exist.
kENETRESETErr	–3251	Unknown.
kECONNABORTEDErr	-3252	Unknown.
kECONNRESETErr	–3253	The connection was reset, possibly due to a problem with security and authentication.

Table B-1 Open Transport result codes (continued)

Result code	Value	Meaning
kENOBUFSErr	–3254	The operation failed because no buffer space was available.
kEISCONNErr	–3255	Reserved.
kENOTCONNErr	–3256	Reserved.
kESHUTDOWNERR	–3257	An operation was aborted because the machine is shutting down. This error code is also used by sockets.
kETOOMANYREFSErr	–3258	Unused error code.
kETIMEDOUTErr	–3259	The requested operation timed out.
kECONNREFUSEDErr	–3260	TCP/IP error code. The port is unreachable (as opposed to the host being unreachable). The positive version of this error code (kECONNREFUSED) is returned by Open Transport as a reason code in a disconnect message, indicating that the other side refused the connection.
kEHOSTDOWNErr	–3263	TCP/IP error. A host address is currently unavailable.
kEHOSTUNREACHErr	–3264	TCP/IP error. A host address cannot be reached.
kEPROTOErr	–3269	A catastrophic error has occurred which probably renders the underlying stream unusable. This error is the same as the kTPROTOErr, but is used where an XTI error code is not appropriate.
kETIMEErr	–3270	An Ioctlcommand has timed out instead of completing normally.
kENOSRErr	–3271	Open Transport cannot allocate enough system resources (usually stream messages) to meet your request.

Table B-1 Open Transport result codes (continued)

Result code	Value	Meaning
kOTClientNotInittedErr	–3279	The client has not called the InitOpenTransport function or the InitOpenTransportUtilities function.
kOTPortHasDiedErr	–3280	Your notifier is sent the event kOTProviderIsClosed and returns this result code if a port that your provider is using is disabled because it was unregistered.
kOTPortWasEjectedErr	–3281	Your notifier is sent the event kOTProviderIsClosed and returns this result code if a port that your provider is using is ejected.
kOTBadConfigurationErr	–3282	Open Transport is attempting to bring up the TCP/IP stack but can't because it's improperly configured in the TCP/IP control panel.
kOTConfigurationChangedErr	–3283	This is an event code that's sent to the client's notifier (the one you register with the OTRegisterAsClient function) when various AppleTalk-related things (like the current zone) change.
kOTUserRequestedErr	–3284	Your notifier is sent the event kOTProviderIsClosed and returns this result code if a port that your provider is using is disabled because the user switched configurations in the TCP/IP or AppleTalk control panels.
kOTPortLostConnection	–3285	Your notifier is sent the event kOTProviderIsClosed and returns this result code if a port that your provider is using is disabled because it lost the connection.

Special Functions

This appendix lists those Open Transport functions that are callable from a hardware interrupt handler or a deferred task handler. This appendix also lists those functions that allocate memory. Any functions not listed in these tables can only be called at system task time.

Functions Callable at Hardware Interrupt Time

Table C-1 lists client functions that you can safely call at hardware interrupt time (from completion routines, VBL tasks, Time Manager tasks, and others), and specifies whether you must call the `OTEnterInterrupt` function before calling the function.

That these functions are safe to call, however, does not imply that they interact predictably with non-interrupt code. For instance, while it is safe to call the `OTAddFirst` function at interrupt time, the actual add operation is not atomic, so calling it at interrupt time could adversely affect non-interrupt code processing the same list. Thus, for each function listed in the table, information is also provided as to whether the operation is atomic.

Table C-1 Functions callable at hardware interrupt time, all ISAs

Function	Needs OTEnterInterrupt?	Atomic
`OTAcquireLock`	no	yes
`OTAddFirst`	no	no
`OTAddLast`	no	no
`OTAllocMem`	no	n/a
`OTAtomicAdd16`	no	yes
`OTAtomicAdd32`	no	yes

Table C-1 Functions callable at hardware interrupt time, all ISAs (continued)

Function	Needs OTEnterInterrupt?	Atomic
`OTAtomicAdd8`	no	yes
`OTAtomicClearBit`	no	yes
`OTAtomicSetBit`	no	yes
`OTAtomicTestBit`	no	yes
`OTCancelSystemTask`	no	n/a
`OTCanLoadLibraries`	yes	no
`OTCanMakeSyncCall`	yes	no
`OTClearLock`	no	yes
`OTCompareAndSwap16`	no	yes
`OTCompareAndSwap32`	no	yes
`OTCompareAndSwap8`	no	yes
`OTCompareAndSwapPtr`	no	yes
`OTCreatePortRef`	no	n/a
`OTDequeue`	no	yes
`OTElapsedMicroseconds`	no	n/a
`OTElapsedMilliseconds`	no	n/a
`OTEnqueue`	no	yes
`OTFindAndRemoveLink`	no	no
`OTFindLink`	no	no
`OTFreeMem`	no	n/a
`OTGetBusTypeFromPortRef`	no	n/a
`OTGetClockTimeInSecs`	no	n/a
`OTGetDeviceTypeFromPortRef`	no	n/a
`OTGetFirst`	no	no
`OTGetIndexedLink`	no	no

Table C-1 Functions callable at hardware interrupt time, all ISAs (continued)

Function	Needs OTEnterInterrupt?	Atomic
`OTGetLast`	no	no
`OTGetSlotFromPortRef`	no	n/a
`OTGetTimeStamp`	no	n/a
`OTIsAtInterruptLevel`	no	n/a
`OTIsInList`	no	no
`OTLIFODequeue`	no	yes
`OTLIFOEnqueue`	no	yes
`OTLIFOStealAndReverseList`	no	yes
`OTLIFOStealList`	no	yes
`OTMemcmp`	no	n/a
`OTMemcpy`	no	n/a
`OTMemmove`	no	n/a
`OTMemset`	no	n/a
`OTMemzero`	no	n/a
`OTRemoveFirst`	no	no
`OTRemoveLast`	noc	no
`OTRemoveLink`	no	no
`OTReverseList`	no	no
`OTScheduleDeferredTask`	yes	n/a
`OTScheduleInterruptTask`	no	n/a
`OTScheduleSystemTask`	yes	n/a
`OTSetBusTypeInPortRef`	no	n/a
`OTSetDeviceTypeInPortRef`	no	n/a
`OTStrCat`	no	n/a
`OTStrCopy`	no	n/a

Table C-1 Functions callable at hardware interrupt time, all ISAs (continued)

Function	Needs OTEnterInterrupt?	Atomic
`OTStrEqual`	no	n/a
`OTStrLength`	no	n/a
`OTSubtractTimeStamps`	no	n/a
`OTTimeStampInMicroseconds`	no	n/a
`OTTimeStampInMilliseconds`	no	n/a
`OTWhoAmI`	no	n/a

Native Functions Callable at Hardware Interrupt Time

Table C-2 lists the functions that are native (providing no mixed-mode glue) and are callable at hardware interrupt time, and also specifies whether you need to call the `OTEnterInterrupt` function before calling the given function. As in Table C-1, information is added to specify whether the operation is atomic.

Note
Some functions listed in Table C-2 are not documented in this manual because they are specific to Open Transport protocol and device drivers. They are listed here to provide a complete reference.

Table C-2 Functions callable at hardware interrupt time, native ISA only

Function	Needs OTEnterInterrupt	Atomic
`OTAddToHashList`	no	no
`OTAllocSharedClientMem`	no	n/a
`OTBufferDataSize`	no	n/a

Table C-2 Functions callable at hardware interrupt time, native ISA only (continued)

Function	Needs OTEnterInterrupt	Atomic
`OTCalculateHashListMemoryNeeds`	no	n/a
`OTCancelTimerTask`	no	n/a
`OTCfigGetInstallFlags`	no	n/a
`OTCfigGetOptionNetbuf`	no	n/a
`OTCfigGetParent`	no	n/a
`OTCfigGetPortRef`	no	n/a
`OTCfigGetProviderName`	no	n/a
`OTCfigIsPort`	no	n/a
`OTCfigNumberOfChildren`	no	n/a
`OTClearBit`	no	yes
`OTEnterGate`	no	yes
`OTFindInHashList`	no	no
`OTFreeSharedClientMem`	no	n/a
`OTInitGate`	no	n/a
`OTIsInHashList`	no	no
`OTLeaveGate`	no	yes
`OTReadBuffer`	no, atomicity is not a factor	n/s
`OTRemoveFromHashList`	no	no
`OTRemoveLinkFromHashList`	no	no
`OTScheduleTimerTask`	no	n/a
`OTSetBit`	no	yes
`OTSetBitRange`	no	yes
`OTSetFirstClearBit`	no	yes
`OTSetLastClearBit`	no	yes

Table C-2 Functions callable at hardware interrupt time, native ISA only (continued)

Function	Needs OTEnterInterrupt	Atomic
`OTTestBit`	no	yes
`StoreIntoNetbuf`	no	n/a
`StoreMsgIntoNetbuf`	no	n/a

Functions Callable From Deferred Tasks

Table C-3 lists Open Transport functions that you can call from a deferred task and specifies whether the call can only be made with a provider in asynchronous mode.

Note
Some functions listed in Table C-3 are not documented in this manual because they are specific to Open Transport protocol and device drivers. They are listed here to provide a complete reference ◆

In general, all endpoint calls can be made asynchronously from a deferred task, with the following limitation.

Table C-3 Functions callable from deferred tasks

Function	Calling restrictions
`OTAccept`	asynchronous only
`OTAckSends`	
`OTAlloc`	
`OTAsynchCreateStream`	

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
`OTAsyncOpenEndpoint`	If opening first endpoint of a configuration, foreground task must be calling `SystemTask` or some other function that calls `SystemTask` (for example, `WaitNextEvent`).
`OTAsyncOpenMapper`	If opening first endpoint of a configuration, foreground task must be calling `SystemTask` or some other function that calls `SystemTask` (for example, `WaitNextEvent`).
`OTAsyncOpenProvider`	If opening first endpoint of a configuration, foreground task must be calling `SystemTask` or some other function that calls `SystemTask` (for example, `WaitNextEvent`).
`OTAsyncStreamOpen`	
`OTAsyncStreamPoll`	
`OTBind`	asynchronous only
`OTCancelReply`	asynchronous only
`OTCancelRequest`	asynchronous only
`OTCancelSynchronousCalls`	
`OTCancelUReply`	asynchronous only
`OTCancelURequest`	asynchronous only
`OTCfigAddChild`	
`OTCfigChangeProviderName`	
`OTCfigCloneConfiguration`	
`OTCfigDeleteConfiguration`	
`OTCfigGetChild`	
`OTCfigNewChild`	
`OTCfigNewConfiguration`	
`OTCfigPopChild`	
`OTCfigPushChild`	

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
OTCfigPushNewSingleChild	
OTCfigPushParent	
OTCfigRemoveChild	
OTCfigSetPath	
OTCfigSetPortRef	
OTCloneConfiguration	
OTCloseMatchingProviders	
OTCloseProvider	
OTCloseProviderByStream	
OTConfiguratorUnloaded	
OTConfigureChildren	
OTConnect	asynchronous only
OTCountDataBytes	
OTCreateConfiguration	
OTCreateStateMachine	
OTCreateTimerTask	
OTDeleteConfigurator	
OTDeleteName	asynchronous only
OTDeleteNameByID	asynchronous only
OTDestroyConfiguration	
OTDestroyStateMachine	
OTDestroyTimerTask	
OTDontAckSends	
OTEnterNotifier	
OTFindOption	

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
OTFindPort	
OTFindPortByRef	
OTFree	
OTGetConfiguratorUserData	
OTGetEndpointInfo	asynchronous only
OTGetEndpointState	
OTGetIndexedPort	
OTGetMessage	asynchronous only
OTGetPriorityMessage	asynchronous only
OTGetProtAddress	asynchronous only
OTGetProviderPortRef	
OTInitHashList	
OTInstallNotifier	
OTIoctl	asynchronous only
OTIsBlocking	
OTIsDependentPort	
OTIsMasterConfigurator	
OTIsSynchronous	
OTLeaveNotifier	
OTListen	asynchronous only
OTLook	
OTLookupName	asynchronous only
OTNewControlMask	
OTNextOption	
OTNotifyAllClients	

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
OTNotifyUser	
OTOptionManagement	asynchronous only
OTPeekMessage	
OTPutBackBuffer	
OTPutBackPartialBuffer	
OTPutMessage	asynchronous only
OTPutPriorityMessage	asynchronous only
OTRcv	asynchronous only
OTRcvConnect	asynchronous only
OTRcvDisconnect	asynchronous only
OTRcvOrderlyDisconnect	asynchronous only
OTRcvReply	asynchronous only
OTRcvRequest	asynchronous only
OTRcvUData	asynchronous only
OTRcvUDErr	asynchronous only
OTRcvUReply	asynchronous only
OTRcvURequest	asynchronous only
OTReadMessage	
OTRegisterName	asynchronous only
OTRegisterPort	
OTRemoveNotifier	
OTRemoveStreamFromProvider	
OTResolveAddress	asynchronous only
OTSetAsynchronous	
OTSetBlocking	

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
OTSetNonBlocking	
OTSetSynchronous	
OTSMCallStateProc	
OTSMComplete	
OTSMCreateControlStream	
OTSMCreateStream	
OTSMGetClientData	
OTSMGetMessage	
OTSMGetState	
OTSMIoctl	
OTSMOpenStream	
OTSMPopCallback	
OTSMPutMessage	
OTSMReturnToCaller	
OTSMSetState	
OTSMWaitForComplete	
OTSnd	asynchronous only
OTSndDisconnect	asynchronous only
OTSndReply	asynchronous only
OTSndRequest	asynchronous only
OTSndUData	asynchronous only
OTSndUReply	asynchronous only
OTSndURequest	asynchronous only
OTStreamClose	
OTStreamGetMessage	asynchronous only

Table C-3 Functions callable from deferred tasks (continued)

Function	Calling restrictions
`OTStreamGetPriorityMessage`	asynchronous only
`OTStreamInstallNotifier`	
`OTStreamIoctl`	asynchronous onl
`OTStreamIsBlocking`	
`OTStreamIsSynchronous`	
`OTStreamPutMessage`	asynchronous only
`OTStreamPutPriorityMessage`	asynchronous only
`OTStreamRead`	asynchronous only
`OTStreamRemoveNotifier`	
`OTStreamSetAsynchronous`	
`OTStreamSetBlocking`	
`OTStreamSetControlMask`	
`OTStreamSetNonBlocking`	
`OTStreamSetSynchronous`	
`OTStreamUseSyncIdleEvents`	
`OTStreamWrite`	asynchronous only
`OTSync`	asynchronous onl
`OTUnbind`	asynchronous only
`OTUnregisterPort`	
`OTUseSyncIdleEvents`	
`OTWhoAmI`	

Functions That Allocate Memory

Table C-4 lists all Open Transport functions that allocate memory on your behalf and therefore require that you call either the InitOpenTransport function or the InitOpenTransportUtilities function before calling them.

Note
Some functions listed in Table C-4 are not documented in this manual because they are specific to Open Transport protocol and device drivers. They are listed here to provide a complete reference.

Table C-4 Functions that allocate memory

Function	Calling restrictions
OTAlloc	needs InitOpenTransport
OTAllocMem	needs InitOpenTransportUtilities
OTAsyncOpenEndpoint	needs InitOpenTransport
OTAsyncOpenMapper	needs InitOpenTransport
OTAsyncOpenProvider	needs InitOpenTransport
OTAsyncStreamOpen	needs InitOpenTransport
OTCreateDeferredTask	needs InitOpenTransportUtilities
OTCreateSystemTask	needs InitOpenTransportUtilities
OTOpenEndpoint	needs InitOpenTransport
OTOpenEndpointOnStream	needs InitOpenTransport
OTOpenMapper	needs InitOpenTransport
OTOpenProvider	needs InitOpenTransport
OTOpenProviderOnStream	needs InitOpenTransport
OTRegisterAsClient	needs InitOpenTransportUtilities

Table C-4 Functions that allocate memory (continued)

Function	Calling restrictions
`OTStreamOpen`	needs `InitOpenTransport`
`OTStreamPipe`	needs `InitOpenTransport`
`OTTransferProviderOwnership`	needs `InitOpenTransport`
`OTUnregisterAsClient`	needs `InitOpenTransportUtilities`
`OTWhoAmI`	needs `InitOpenTransport`

XTI Option Summary

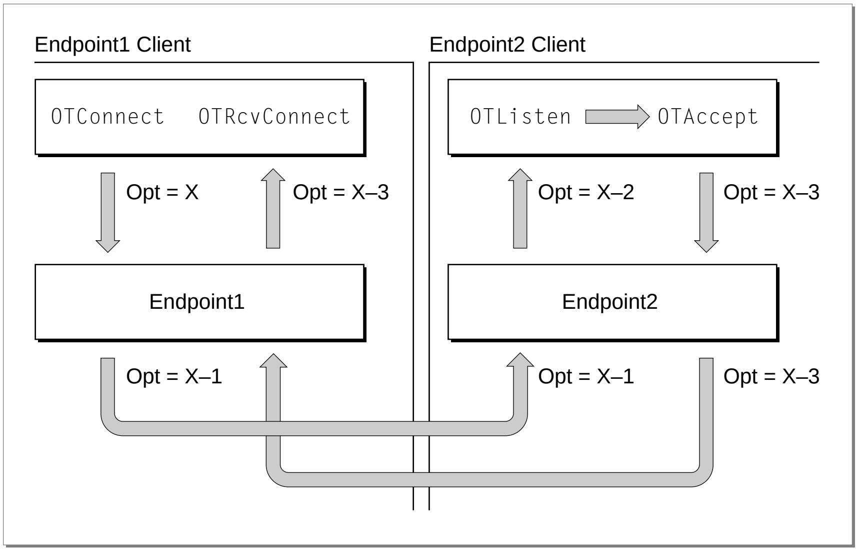
This appendix summarizes information about types of options and option negotiation rules.

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 D-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 D-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 D-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 D-1 shows which Open Transport functions used to specify options can accept association-related options and which can accept both types of options for input and output parameters.

Table D-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

Table D-1 Open Transport endpoint functions and the types of options they accept (continued)

Function	Input parameter	Output parameter
`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. In some cases, nonprivileged clients can read the value of a privileged option. Currently, the Mac OS does not have a privileged model, so this distinction is irrelevant.

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. 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.

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 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 on.
- 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 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.

Options Negotiation Rules

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. Those options that were successfully negotiated retain their new values. There is no undo mechanism.

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. If the option is not supported, the status field is set to T_NOTSUPPORT.

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 with a value that is currently set. 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 Mac OS 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" (page 813). 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 7-2 (page 169) 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.

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

The following sections describe how you retrieve option values for different kinds of endpoints.

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 D-1 (page 808). 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.

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.

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.

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 Anattribute that identifies the type of address format used for an Open Transport 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 transaction.**

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) An extension of ADSP to provide authentication and encryption.

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 applications each time certain network-related events occur, such as opening or closing an AppleTalk driver. Any applications that rely on the ATQ events, other than the miscellaneous events supported by Open Transport, must use AppleTalk backward compatibility to handle them in the classic AppleTalk manner. See also **miscellaneous event.**

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 requires each peer to agree on a clock rate before communicating.

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, completion event.**

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.

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. 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 linel.

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 nonblocking, any function that might have to wait, returns immediately with an error result. See also **asynchronous**.

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 Mac OS 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.

client A protocol that uses the services of an underlying protocol. For example, ADSP is a client of DDP. Also used to refer to application level code, which is a client of the network provider.

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, asynchronous.**

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 for DDP packets. This is used by DDP endpoints to filter incoming and outgoing data.

Data Terminal Ready (DTR) signal A signal indicates that the computer is ready to communicate. Deasserting this signal causes the sender to suspend transmission.

default port The port that Open Transport uses when a specific port is not indicated. The LocalTalk default port is specified in the AppleTalk control panel.

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 DDP 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. The name is not abbreviated, for example `www.apple.com` as opposed to `www`.

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. Also, serial handshaking is used for flow control. 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 address and one or more domain names.

interface function A function that Open Transport passes through to the underlying software modules that implement networking protocols. See also **utility function.**

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”) . Also called **IP address.**

Internet Protocol (IP) The basic datagram-delivery protocol in 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 gateway, 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.

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 A provider's attribute that determines whether provider functions execute synchronously or asynchronously, whether functions can wait to send or receive data, and whether data is copied.

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 A 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 AppleTalk node ID 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 reference 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.

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 services.

name registry A register of hardware and software configuration information for Power Macintosh computers.

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. See also **type** and **zone**.

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 function `OTSetAddressFromNBPEntity` inserts 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 on an AppleTalk network.

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 solely between the client and its endpoint provider. Such options contain no information for the remote client. Compare with **association-related options.**

nonblocking A mode of operation in which a provider will 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 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.

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 a default port. For example, 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, for example, "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: 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 contains information about a 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 values can be changed only by privileged clients, although it is sometimes possible for nonprivileged clients to read the value of a privileged option. The Mac OS does not distinguish between priviledged and nonpriviledged clients; so this distinction is irrelevant to Open Transport.

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 values you can read but not change.

receive queue An ADSP 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 packet's address the most efficient route for the packet. A packet 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 An ADSP 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 protocol stack. 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 to signal data. Compare with **mark**.

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 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. This 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.**

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).

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 divides data transmission into transactions. 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 notdivide the data up into transactions. 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 identifies the type of service that the entity provides, for example, "Mailbox" for an electronic mailbox on a server. See also **name** and **zone**.

UDP User Datagram Protocol.

User Datagram Protocol (UDP) A connectionless datagram protocol that provides port demultiplexing and data checksums.; part of the TCP/IP protocol family.

utility function A function that is implemented by Open Transport itself. See also **interface function.**

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. See also **name** and **type**.

Zone Information Protocol (ZIP) An AppleTalk protocol that maps network numbers to zone names for all networks belonging to an AppleTalk internet.

Index

THE APPLE PUBLISHING SYSTEM

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
Joanna Bujes, Robin Joly

DEVELOPMENTAL EDITORS
Donna S. Lee

ILLUSTRATORS
Karin Stroud, Dave Arrigoni

PRODUCTION EDITORS
Glen Frank, Gerri Gray,

PROJECT MANAGER
Tony Francis

Special thanks to Quinn, Eric Ockholm, Mike Quinn, Don Coolidge, Lauren Sherman, Steve Ussery, Tony Wingo, Rich Kubota, and Vinnie Moscaritolo.

Acknowledgments to Jose Carreon, Michel Guittet, Garry Hornbuckle, Jeri Sonnenberg, and the entire Open Transport engineering team.