SERIES EDITOR **Macintosh Inside Out** SCOTT KNASTER

Programming WITH AppleTalk®

MICHAEL PEIRCE

Programming with AppleTalk®

JACK Lee

[illegible]

Programming with AppleTalk®

Michael Peirce

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Library of Congress Cataloging-in-Publication Data

Peirce, Michael, 1961-
Programming with AppleTalk / Michael Peirce.
p. cm. — (Macintosh inside out)
Includes index.
ISBN 0-201-57780-1
1. Macintosh (Computer)—Programming. 2. AppleTalk. 3. Local area networks (Computer networks) I. Title. II. Series.
QA76.8.M3P45 1991
005.7'1369—dc20 91-4321
CIP

Sponsoring Editor: Rachel Guichard
Technical Reviewer: Peter Helme
Cover Design: Ronn Campisi
Set in 10.5-point Palatino by Total Concept Associates

1 2 3 4 5 6 7 8 9 -MW- 95 94 93 92 91

First Printing, August 1991

For Kathleen, the Spud, and the Roo

Contents

Foreword by Scott Knaster

What a difference "always" makes! Folks who make personal computers offer lots of options for their buyers: different kinds of monitors, graphics cards, printers, keyboards, disk drives, and so on. This provides great flexibility for customers. They can figure out exactly what they need and they don't have to spend money on stuff that doesn't fit their requirements.

Usually, though, computer technology goes faster than the ability of people to learn about it. Neat new Things are invented before computer buyers can understand what they're good for. In cases like this, most buyers won't be interested in the new, unfamiliar Thing, because they can't see why it's useful to them. When nobody buys the new Thing, software developers are reluctant to write software that takes advantage of it. That's a vicious circle: Nobody buys it, so nobody writes software for it, so nobody buys it.

How do you break the circle? The boldest way is by building the Thing into the computer in the first place. Steve Jobs and the folks who made the Macintosh understood this principle and were willing to take the risks that it forces. The Macintosh designers believed in the mouse and bitmapped graphics as fundamental parts of the computer, so they made sure that nobody could ever get a Macintosh without those vital pieces.

Note that building breakthrough features into a computer isn't always an obvious, easy win. Every little feature that's added to a computer costs money and makes the retail price go up, which causes a few more people to think twice (or change their minds) about buying the computer. Building in breakthroughs is not a strategy for the timid.

One of the Macintosh's niftiest built-in breakthroughs was the AppleTalk network system and the LocalTalk connector on the back (originally, both the network and the wire were called AppleTalk). For the first time, computer buyers didn't have to think about whether they wanted to spend extra money to hook their computers together (which is a good thing, because there wasn't much for them to do after they were hooked together way back in 1984).

Once Apple broke the spell by shipping the Macintosh, a computer which included built-in network hardware and software, the vicious circle started to collapse. Helped by the LaserWriter's success, smart folks started adding goodies like file servers and electronic mail to the network. As the state of the art progressed, connections between networks appeared, as well as faster transport systems for heavy-duty applications.

Today, the network brings us file sharing, program linking, and other magic. In this book, Michael Peirce shows you how you can take advantage of this doorway to the world that's inside every Macintosh. Michael gives you access to his years of experience in exploring the AppleTalk connections between Macintoshes. If you supply the creative spark, you can use Michael's wise guidance to help you follow the Macintosh's networking vision.

Scott Knaster
Macintosh Inside Out Series Editor

Acknowledgments

Writing a book is a big job and there is no way I could have finished this book without the support of many folks.

My most important supporter is my wonderful wife Kathleen. Without her support, this project would have never taken place. Thank you, Kathleen, for believing in me and sharing my dreams.

An important catalyst for this book was Tony Meadow. It was over pasta at Florentine's that this book first took shape. Thank you Tony for putting me in touch with the good people at Addison-Wesley.

The people at Addison-Wesley have been a pleasure to work with. Their professionalism and drive to make this the best book possible is appreciated. Rachel Guichard got the ball rolling and took care of so many details. Joanne Clapp Fullagar provided much needed advice, guidance, and editing. Valerie Haynes Perry did such a great job of copy editing that this book may now be readable, and Kathy Traynor was able to take things from mere words and make it a real book.

Another important person was Pete Helme from Apple Computer who provided value technical feedback and caught many a subtle gaffe on my part.

I'd also like to thank Apple Computer for sneaking such a great networking system into such a little box.

And of course, I must thank my cats, Sushi and Schnitzel, for keeping my lap warm and entertaining me as I wrote this book.

Introduction

This book is about writing programs that use AppleTalk, the networking system built into every Macintosh computer made by Apple Computer. Because AppleTalk is built-in, programmers can rely on it for program-to-program communication over a network in the same way they can rely on QuickDraw to handle their graphics needs.

▶ Who This Book Is For

This book is for programmers who want to learn to use AppleTalk in their programs written for the Macintosh. It assumes an understanding of programming the Macintosh, though not necessarily any previous network programming experience.

Programmers reading this book should be familiar with Pascal. The examples used in this book were written using the MPW Pascal compiler from Apple Computer and have been compiled using MPW Pascal version 3.1. The examples also work with little or no modification with THINK Pascal from Symantec. However, extensive knowledge of Pascal is not required; the Pascal code found in this book is easily transliterated into other similar languages such as C or Modula 2.

This book focuses on the AppleTalk protocols that are most practical for use with Macintosh application programs. The protocols that allow you to find network entities and easily transfer data back and forth over the network are covered in a fair amount of detail. Other protocols that are for special purposes not of general interest to the application programmer are briefly described, but not covered in detail.

This books also covers a number of network related areas. How to build a Chooser-based interface is discussed, as is the construction of a resident INIT. Topics such as completion routines and the use of synchronous and asynchronous device calls that are important to AppleTalk programming can also be applied to non-AppleTalk-related programming.

▶ Who This Book Is Not For

This book is not for those who are trying to write their own implementation of the AppleTalk protocols. It assumes that the reader will be using the standard programming interfaces provided by Apple for the Macintosh computer. People who are interested in these low-level protocol issues are referred to *Inside AppleTalk*, published by Addison-Wesley. It provides a detailed description of exactly how all the protocols operate at a fundamental level.

This book is also not specifically for those trying to set up and maintain an AppleTalk network. Although this book may provide some insight that can be helpful in administering a network, this book is not about AppleTalk network management.

▶ Organization of This Book

Programming with AppleTalk book is organized into three parts. The first part provides an introduction to the AppleTalk system and how its pieces fit together. It also details each protocol found in AppleTalk and discusses why you may or may not want to use that protocol in your own programs.

The second part digs into the details of using AppleTalk. Various techniques are discussed and issues explored that are important when designing your AppleTalk-based programs.

Each of the major protocols used by application programs is discussed in detail in the second part. Each call in those protocols is explained and you are shown how to use the calls. Detailed, field-by-field explanations of each of the important calls found in the protocols are also included in this part.

The final part of the book is an in-depth discussion of three working programs—NameTool, Remote SysInfo, and Checkers—that use AppleTalk. Although very different types of programs, they each illustrate important aspects of AppleTalk programming. They can be used as a starting point for your own projects.

▶ Summary

This introduction provided some guidance about who would most likely benefit from reading this book and who might not. It also discussed what type of background is required to fully take advantage of the material found in this book.

Chapter 1 delves into the concepts and terms you will need to become familiar with in order to understand Macintosh networking.

PART ONE

▶ Basic Networking on the Macintosh

Part One of this book gets you started with the fundamentals of network programming on the Macintosh. It explains the basic concepts of networking in general as well as discusses each of the protocols provided in AppleTalk and what they are used for. When you finish Part One you should have a basic understanding of the issues involved in programming AppleTalk on the Macintosh. This part consists of two chapters.

- Chapter 1 introduces some basic concepts in network programming. It focuses on the AppleTalk protocol stack and how it can be fit into the International Standards Organization Open Systems Interconnect (ISO-OSI) seven-layer model.
- Chapter 2 discusses a variety of programming issues related to AppleTalk programming. It covers which protocol to use in various circumstances. It also covers the issues involved in internetworking and using the preferred AppleTalk interfaces.

1 ▶ Networking on the Macintosh

This chapter introduces you to some basics of networking and how networks function. It shows how AppleTalk networking fits in with other communications systems on the Macintosh and puts AppleTalk networking in the context of other computer networking systems. The ISO-OSI reference model is discussed as a model for a general networking system and the chapter shows how AppleTalk can be placed into the ISO-OSI framework.

▶ The Macintosh Communications Environment

The Macintosh computer has a rich variety of communications systems available as part of its standard system software. Three general areas exist: serial communications, Apple events, and AppleTalk networking.

Serial communications allows the Macintosh to send streams of data in and out over its serial lines. This data moves at a relatively low speed and is most commonly used to communicate with other computers, printers, or modems, often with the Macintosh emulating a terminal. This simple model has recently been enhanced with the addition of the Communications Toolbox (CTB) so that serial data can be transmitted using serial lines other than just the two that are built-in.

Apple events, available with System 7.0 and later systems, provides a standard way for Macintosh applications to send control information and share data. In most circumstances this takes place within one machine, but Apple events can be sent over the network. Underlying Apple events can be found in the Program to Program Communication (PPC) Toolbox.

This provides a standard way to convey relatively small amounts of data between programs.

AppleTalk networking provides a rich and varied set of protocols that allow Macintoshes connected by network hardware to share data. This type of communication is much more general than the two previous types, so much so that both serial communications and Apple events can be partially implemented on top of AppleTalk.

In the case of serial communications, the Communications Toolbox provides a tool that lets two Macintoshes connected by an AppleTalk network exchange data as if they were connected via serial lines. This tool, called the ADSP tool, is built on top of the AppleTalk Data Stream Protocol (ADSP) and is part of the standard set of CTB tools.

In the case of Apple events, all Apple events that are sent between different Macintoshes are conveyed using the AppleTalk network.

Each of these communications environments is appropriate for the tasks for which it was built. There is some overlap between them of course, but close examination of the problem to be solved can tell you which type of communications system is most appropriate for your task.

▶ AppleTalk Networking

A computer network can be defined as a group of computers all interconnected to allow easy exchange of information with each other. Networks are used to share resources beyond those available to just a single machine. AppleTalk is such a system. It physically interconnects Macintosh computers, as well as other devices such as printers, file servers, shared modems, and a wide variety of other machines, and allows them to exchange information with each other.

AppleTalk is made up of a set of hardware protocols and software protocols that allow this interconnectivity to take place. In the early days of AppleTalk, it was not much more than a nice way to connect Macintosh computers to LaserWriter printers. Today, though still a fine way to communicate with printers, AppleTalk has grown to support a very diverse set of interactions. These range from file sharing and file transfer to electronic mail and teleconferencing. New uses for AppleTalk continue to be introduced all the time.

▶ The AppleTalk Protocol Stack

AppleTalk is made up of a wide variety of interrelated protocols. Each protocol builds on the others with each providing a unique set of capabilities.

In order to organize groups of network protocols and bring some structure to the way we talk about them, the International Standards Organization came up with what they call the Open Systems Interconnect reference model (ISO-OSI). This organizes network protocols into seven layers: physical, data link, network, transport, session, presentation, and application. Figure 1-1 shows a diagram of the ISO-OSI reference model.

In a networking system that conforms completely to the ISO-OSI reference model, each protocol fits into exactly one layer and only communicates with layers immediately above and below it. In practice, many network systems only roughly conform to the model, but it is still helpful to view these systems using ISO-OSI reference model terms.

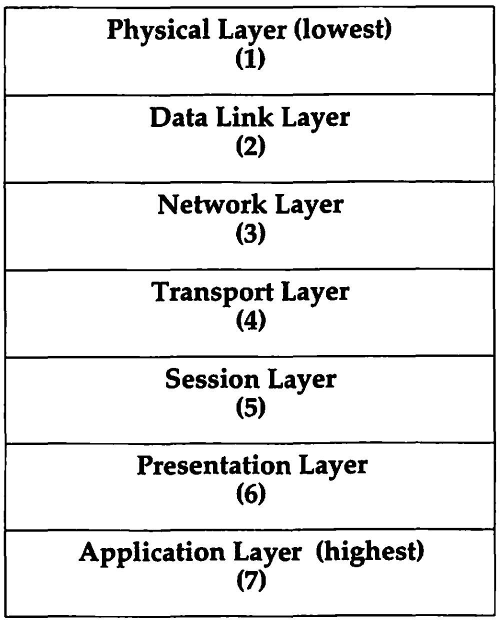


Figure 1-1. The ISO-OSI seven-layer reference model

AppleTalk actually conforms rather well to the ISO-OSI reference model. Some of the protocols fit into more than one layer and many of the protocols directly make use of layers more than one position away. Still, the ISO-OSI reference model can provide a good way to talk about the relationships between the various AppleTalk protocols.

Figure 1-2 shows how the AppleTalk protocols can be mapped onto the ISO-OSI reference model.

Physical Layer (lowest)	Ethernet Hardware	LocalTalk Hardware		Token Ring Hardware
Data Link Layer	EtherTalk Link Access Protocol (ELAP)	LocalTalk Link Access Protocol (LLAP)		TokenTalk Link Access Protocol (TLAP)
Network Layer		Datagram Delivery Protocol (DDP)		
Transport Layer	AppleTalk Transaction Protocol (ATP)	AppleTalk Echo Protocol (AEP)	Routing Table Maintenance Protocol (RTMP)	Name Binding Protocol (NBP)
Session Layer	AppleTalk Session Protocol (ASP)	Zone Infomation Protocol (ZIP)	AppleTalk Data Stream Protocol (ADSP)	Printer Access Protocol (PAP)
Presentation Layer		AppleTalk Filing Protocol (AFP)		
Application Layer (highest)		Your Application		

Figure 1-2. AppleTalk protocols in the ISO-OSI reference model

▶ The Physical Layer

The lowest possible layer in the ISO-OSI reference model is the physical layer. It is responsible for transmitting raw data across the various physical wires, optic fibers, radio links, and so forth. It is concerned with details about mechanical and electrical interfaces such as how to transmit a one or a zero.

In AppleTalk networks, this layer encompasses the LocalTalk, Ethernet, token ring, or other hardware.

▶ The Data Link Layer

The data link layer is built upon the physical layer. Its responsibility is to take the raw physical layer and provide a way to send packets of data back and forth. This layer deals with such issues as error detection, collision detection, and basically taking care of all the details involved with accessing a specific physical layer.

On AppleTalk networks, the data link layer consists of the various Link Access Protocols, such as EtherTalk Link Access Protocol (ELAP), LocalTalk Link Access Protocol (LLAP), and TokenTalk Link Access Protocol (TLAP).

▶ The Network Layer

The network layer is built upon the data link layer. It is responsible for routing packets of data from node to node. This layer deals with such issues as packet generation and management.

On AppleTalk networks, the network layer consists of the Datagram Delivery Protocol (DDP).

▶ The Transport Layer

The transport layer is built upon the network layer. It is responsible for conveying reliable information across the network and providing this service to the session layer.

On AppleTalk networks, the transport layer primarily consists of the AppleTalk Transaction Protocol (ATP), although the Routing Table Maintenance Protocol (RTMP), AppleTalk Echo Protocol (AEP), and Name Binding Protocol (NBP) are often lumped into this layer.

▶ The Session Layer

The session layer is built upon the transport layer. It is responsible for establishing and maintaining high-level sessions. These sessions provide a higher level of service than the transport layer.

On AppleTalk networks, the session layer consists of the AppleTalk Session Protocol (ASP), Zone Information Protocol (ZIP), AppleTalk Data Stream Protocol (ADSP), and Printer Access Protocol (PAP).

Remember that the ISO-OSI reference model is simply a guide. For efficiency, some protocols access layers farther down the stack than a single layer. For example, ADSP makes direct use of DDP on the network layer rather than using a protocol found on the transport layer.

▶ The Presentation Layer

The presentation layer is built on the session layer. It provides common services for the application layer. These services could well reside in the application layer, but are common or useful enough that they are provided as part of the overall networking system.

On AppleTalk networks, the presentation layer contains the AppleTalk Filing Protocol (AFP).

▶ The Application Layer

The Application layer is built on top of all the other protocols. It contains a wide variety of application-specific protocols built up by you, the application programmer.

On AppleTalk networks, the application layer consists of many application-specific protocols. Each network-based product makes use of some of the protocols found in AppleTalk and imposes its own structure on them, forming its own special protocol.

▶ Summary

This chapter discussed the concept of computer networks. It described how AppleTalk networking fits into the overall scheme of Macintosh communication software. It also described the ISO-OSI reference model and how the AppleTalk protocols can fit into this model.

Chapter 2 discusses AppleTalk programming issues involving the various protocols.

2 ▶ AppleTalk Programming Issues

In this chapter you will learn about some of the issues involved when you program AppleTalk on the Macintosh. A quick synopsis of the available AppleTalk protocols is provided. This can help you to decide which protocol to use for various tasks. Other areas discussed include using the so-called preferred interface and some general networking issues such as internetworking and its impact on your code, as well as the various transport media available.

▶ Using the Various AppleTalk Protocols

AppleTalk offers a wide variety of protocols. These range from low-level protocols that provide basic services, such as the various Link Access Protocols, to high-level protocols that provide advanced services, such as AppleTalk Filing Protocol. Fortunately, most of the AppleTalk protocols are for very specialized uses and need not be used by most programmers.

This book provides a detailed look at the major AppleTalk protocols used by the vast majority of AppleTalk applications programs. AppleTalk contains rich and varied protocols and some of the less widely used protocols may also be of interest to you. For this reason, the following section provides a broad overview of all the AppleTalk protocols. Some of the less frequently used protocols that this book does not cover in great detail are provided with references to aid you in finding further information about them.

▶ Link Access Protocols

AppleTalk provides a variety of Link Access Protocols. These provide access and management of the various hardware interfaces.

Originally, LocalTalk was the only supported network hardware in AppleTalk and it was managed by the LocalTalk Link Access Protocol (LLAP). Because LocalTalk hardware is built into all Macintoshes and LaserWriters, LLAP is the most widely used Link Access Protocol.

By the Way ▶

Originally, Apple used the word AppleTalk to refer to both the entire AppleTalk protocol suite and the physical connections we now call LocalTalk. This was unambiguous when LocalTalk was the only Link Access Protocol used by AppleTalk, but once other Link Access Protocols were provided, primarily Ethernet, things became confusing. That's when Apple adopted the AppleTalk nomenclature to specifically refer to the protocol suite. Apple then came up with the term LocalTalk to describe the built-in networking hardware. This allowed them to use EtherTalk to refer to AppleTalk protocols running on Ethernet hardware and TokenTalk to refer to AppleTalk protocols running on token ring hardware.

Two other Link Access Protocols are also in fairly wide use on AppleTalk networks. They are the Ethernet Link Access Protocol (ELAP) and TokenTalk Link Access Protocol (TLAP). ELAP provides access to Ethernet based networks, whereas TLAP provides access to token ring based networks.

The various Link Access Protocols provide very low-level access to the AppleTalk network that is seldom needed. Also, because the different Link Access Protocols may or may not be available on a given machine, very few programs deal with them directly.

▶ Datagram Delivery Protocol

Datagram Delivery Protocol (DDP) is the low-level workhorse protocol in AppleTalk. It provides socket-to-socket, best effort delivery of packets on the network. It uses whichever Link Access Protocol the user has selected.

DDP is used by all higher-level protocols in AppleTalk to deliver their data across the network. Because DDP only provides best effort data delivery, any protocol or program that makes direct use of DDP should expect to lose packets on occasion and should be prepared to handle this loss.

DDP can be an appropriate choice for some programs to use directly. However, other higher-level protocols provide more services and make it much easier to reliably move data between programs. DDP is discussed briefly in this book.

▶ Routing Table Maintenance Protocol

Routing Table Maintenance Protocol (RTMP) provides a way for internet routers to manage their routing tables. These routing tables allow the internet routes to forward datagrams between different networks. RTMP is not normally used by application programs.

▶ AppleTalk Echo Protocol

AppleTalk Echo Protocol (AEP) provides one simple service: the ability to send a single DDP packet to a node and have it echoed back. This service is used to detect if a node is online and to measure round-trip packet transmission times.

To detect if a remote node exists, AEP can be used to ask a remote node to echo back a DDP packet. If this packet comes back, the node exists. Note, however, because DDP is used to transport the data, the packets may be dropped. Repeat attempts should be made to establish contact before you conclude that the remote node is not online.

To measure round-trip transmission times, a packet is sent using DDP. It is then received back after measuring the time between the two transmissions. This information can be used to tune the use of the network. Of course, this method requires repetition to determine a good average round-trip time.

▶ AppleTalk Transaction Protocol

AppleTalk Transaction Protocol (ATP) provides a reliable way to exchange data between two sockets. Its basic model of operation is the transaction, that is, a request followed by a response.

ATP transactions are a simple and efficient means of transporting relatively small amounts of data across the network. ATP is extensively discussed in this book.

▶ Name Binding Protocol

Name Binding Protocol (NBP) is another of AppleTalk's workhorse protocols. It is used by almost all programs using AppleTalk because it provides the means for finding services on AppleTalk.

AppleTalk provides a very flexible address assignment scheme. Addresses are assigned dynamically to nodes as they join the network. This means that services cannot be assigned a fixed address that programs can use to find them. This is where NBP comes in. It allows programs to assign names to themselves that other programs can look up and then get a full address from that name.

Name Binding Protocol should be used by your programs to advertise their existence so that other programs can make contact with them. NBP should also be used to find remote services.

Name Binding Protocol is extensively discussed in this book.

▶ Zone Information Protocol

Zone Information Protocol (ZIP) is used to maintain the zone name-to-network number mappings on internet routers. ZIP also provides a way for non-router nodes to get the zone-to-network mapping information. Chapter 6 discusses the use of ZIP to obtain zone information from routers.

▶ AppleTalk Session Protocol

AppleTalk Session Protocol (ASP) provides an asymmetric transport mechanism between client programs and a server program. Sessions are always initiated by the clients. These clients initiate requests of the server and the server is responsible for servicing those requests.

ASP is appropriate for basic client and server based networking. Unfortunately, only the client side of ASP is implemented on the Macintosh. If someone wishes to write a server for ASP, they are required to implement it themselves. Because of this, ASP is beyond the scope of this book.

▶ Printer Access Protocol

Printer Access Protocol (PAP) is an asymmetric protocol that provides access to printer servers such as the Apple LaserWriter.

PAP is not used beyond the scope of printer access and is therefore beyond the scope of this book.

▶ AppleTalk Data Stream Protocol

AppleTalk Data Stream Protocol (ADSP) provides a symmetric, full-duplex, reliable stream of data between two programs.

ADSP allows two programs to exchange large amounts of data very efficiently. It allows the programs to treat the data streams as either continuous streams of data or logically divided messages that can be individually interpreted. It also provides an out-of-band attention signal. This attention signal allows control information to pass without disrupting the primary data streams.

ADSP is discussed extensively in this book.

▶ AppleTalk Filing Protocol

AppleTalk Filing Protocol (AFP) provides a way to access remote AppleShare file systems. It is an asymmetric protocol based on ASP.

Normal access to remote AppleShare servers is accomplished by using the file system and does not require direct use of AFP. Also, only the AFP client software is provided on the Macintosh. AFP is not discussed further in this book.

▶ Internetworking

AppleTalk networks are often interconnected to form larger networks known as *internetworks* or simply *internets.* This is done using a variety of devices. These include devices known as repeaters, routers, and gateways. Each approach provides a different set of services and performance characteristics.

Generally, the use of internetworking is transparent to your networking program, but because they affect network performance, care needs to be taken to allow for their impact. Transmission delays and lengthy timeouts are the main consideration.

▶ Repeaters

The simplest forms of internetworking are *repeaters*. Repeaters provide a way to extend wiring beyond its normal limitations. Repeaters simply receive data transmissions and retransmit them onto another wire. They do not modify the data in any way as they retransmit it.

Logically, repeaters are transparent to the network software. They do introduce a very minor transmission delay, but in practice, it is virtually insignificant.

▶ Routers

Routers, also known as *bridges*, connect two or more segments of a network together. Unlike repeaters, they store and forward data. This means they receive a complete data packet before retransmitting it out the other side.

In AppleTalk networks, routers define zone names for their constituent networks. They communicate among themselves to keep this information up-to-date using the Routing Table Maintenance Protocol (RTMP).

Routers come in a variety of configurations. Some routers are made up of two half-routers, that is, two devices that cooperate together to provide a full router. Half-routers are connected by some type of transport medium, such as a phone line or a microwave link. These connections are often slower than those found at either end.

Other types of routers can be connected to a large number of networks that use a variety of transport media. A common type of router in AppleTalk networks is one that is connected to an Ethernet network and routes data to a number of LocalTalk networks.

Routers often process the data that passes through them. Some routers can be configured to isolate certain types of traffic. This can lead to significantly improved network performance by keeping local data on a local network instead of taking up bandwidth throughout a large networking system.

Routers can introduce significant transmission delays, especially if they are half-routers with a slow intermediate transport media (such as a phone line). These situations can often lead to timeouts occurring if your timeout values are set too low.

Logically, routers are virtually transparent to the network software (unless you need to talk to it to get zone information). But care must be taken to consider the wide variety of network topologies that users may be operating on when designing your software.

▶ Gateways

Gateways translate the protocols of one network into the protocols of the other network. Gateways provide a way to connect very different networks together. For example, gateways are available to connect AppleTalk networks with DECnet networks. Each of these networks uses a completely different set of protocols, but there is some commonality between them.

Gateways often do an incomplete job of translation because all the services of one networking system are rarely found on another networking system. They also can introduce significant delays as translation must be performed rather than simple data retransmission.

Using protocol gateways often requires special attention in your programs. Various gateways can impose important restraints on your program because of the way protocol conversion is done in the gateway.

▶ The Preferred AppleTalk Interface

The original Pascal interface supplied for AppleTalk, and discussed in *Inside Macintosh*, Volume II, describes what is now known as the *alternate interface*. This interface used a parameter block containing all relevant information for the call. It also used a flag to signal the synchronous or asynchronous version of the call. (Chapter 3 discusses synchronous and asynchronous operations in detail.)

In the synchronous version of the call, the program was blocked until the specified operation completed. In the asynchronous version of the call, program control returned immediately and completion of the operation was signaled by the posting of what was known as a network event to the event queue.

By the Way ▶ Network events are similar to other events, such as mouse down or update events, except that they signal some network activity.

This scheme using network events had a number of problems, among them the fact that this could only work for applications, not other types of code such as VBL tasks. Also, it was not MultiFinder friendly as events were not guaranteed to be posted to the originating application.

To remedy this situation a new interface was introduced in *Inside Macintosh*, Volume V, called the *preferred interface*. The preferred interface was modeled after the Device Manager. Much like the alternate interface, the preferred interface uses a parameter block and a flag to denote synchronous or asynchronous operation. The preferred interface lets you use a completion routine rather than handle a network event.

With the new preferred interface, programming AppleTalk became much more like programming other toolbox services such as the File Manager or the Serial Manager. Similar techniques could be used and non-application access to asynchronous operations became available. In short, the preferred interface made programming AppleTalk much more flexible than it was in the days of the alternate interface. This book uses the preferred interface.

▶ AppleTalk Transport Media

In the early days of AppleTalk, discussing the transport media available for AppleTalk was very straightforward because there was only one type of transport media available: LocalTalk. Today AppleTalk can use a rich variety of transport media including LocalTalk, Ethernet, token ring, and others. Each of these media brings its own mix of speed, costs, and reliability.

▶ LocalTalk

LocalTalk is the most common AppleTalk transport medium simply because LocalTalk hardware is built into every Macintosh ever manufactured by Apple Computer. Thus, it provides the lowest common denominator for comparing AppleTalk networks.

LocalTalk hardware can transfer data at a rate of 230.4 kilobits per second over a twisted-pair wire. This translates to 28.8 kilobytes per second and, with overhead, you can expect to transfer approximately 25 kilobytes of data per second over an otherwise unused LocalTalk cable. Of course, networks are shared resources and often have other traffic vying for their capacity, so actual data transfer speeds can be even lower.

When many users are accessing a LocalTalk network, some mechanism must be used to arbitrate the usage of the cable. Only one machine can be transmitting at any one time. To minimize these access collisions, LocalTalk uses what's known as CSMA/CA or Carrier Sense Multiple Access with Collision Avoidance.

CSMA/CA is a sophisticated technique that attempts to minimize the chances of two machines trying to transmit data over the same wire at the same time. It works by sensing if the wire is already in use. If so, it defers to the transmission already in progress. It then waits a minimum amount of time after the wire is quiet plus a small random amount of additional time before attempting to begin its transmission.

Once a transmission has begun, LocalTalk cannot sense if a collision has occurred. Rather, it sends a very short *handshake* across the wire and from this it can infer that a collision has occurred. If there has been no collision, the entire transmission is sent. In practice, little data is lost due to collisions.

The data carrying capacity of LocalTalk is limited compared with most other network transport media. Because it trades off speed for low cost, it can be built into a wide variety of relatively low-cost computers, printers, and other peripherals. And even at its somewhat slow speed, it has proven to be a capable transport medium for small networks and a feeder network for larger internets.

▶ Ethernet

AppleTalk running on Ethernet accounts for a growing number of AppleTalk networks. It provides a significantly higher performance network medium than does LocalTalk.

Ethernet networks can transfer data at a rate of 10 megabits per second over a coaxial or twisted-pair wire. This translates to 1280K per second and with overhead, well over 1 megabyte per second.

When many users are accessing an Ethernet network, some mechanism must be used to arbitrate the usage of the cable. Only one machine can be transmitting at any one time. To minimize these access collisions, Ethernet uses what's known as CSMA/CD or Carrier Sense Multiple Access with Collision Detection.

Like CSMA/CA, CSMA/CD is also a sophisticated technique that attempts to minimize the chances of two machines trying to transmit data over the same wire at the same time. It works by sensing if the wire is already in use. If so, it defers to the transmission already in progress. It then waits a random amount of time before attempting to begin its transmission.

In contrast to CSMA/CA, once a transmission has begun, CSMA/CD can sense if a collision has occurred and the transmitters both back off and try again after another random wait.

The data carrying capacity of Ethernet is quite adequate for a wide variety of data transmission tasks. Because there is often spare capacity on an Ethernet, it is well suited for providing a large number of connections or acting as a backbone for numerous LocalTalk networks.

Ethernet provides roughly an order of magnitude better performance for roughly an order of magnitude increase in cost, although these economics are changing as Ethernet hardware continues to drop in price.

▶ Token Ring

AppleTalk running on token ring accounts for a small number of AppleTalk networks. Most of these networks are in place to provide connectivity with token ring based networks.

Token ring networks can transfer data at a rate of 4 megabits per second over a coaxial or twisted-pair wire. This translates to 512 megabits per second and with overhead, just under 1/2 megabyte per second.

Token rings operate very differently than either LocalTalk or Ethernet. Token rings are constructed in a ring shape and continually pass data around that ring. A token is passed around the ring when it is idle, and when a machine wants to transmit data, it must first take the token from the network. In this way, it precludes other machines on that same ring from gaining the token and trying to transmit. When the transmission is complete, the token is returned to the ring and other machines can then take it and have their turn at transmitting.

The data carrying capacity of token ring is adequate for a wide variety of data transmission tasks. It isn't quite as fast as a 10Mbps Ethernet, but it is still much faster than LocalTalk. It is well suited to speedy data transmission.

▶ Summary

This chapter discussed the various protocols that make up AppleTalk. Also discussed in this chapter were internetworking issues and how various transport media used by AppleTalk affect your programs.

Information on AppleTalk protocols may be found in several chapters of several volumes of *Inside Macintosh* as well as in other books. See the following references:

ADSP	*Inside Macintosh,* Volume II; *Inside Macintosh,* Volume V; and *Inside AppleTalk.*
AEP	*Inside Macintosh,* Volume V and *Inside AppleTalk.*
AFP	*Inside Macintosh,* Volume V and *Inside AppleTalk.*
ASP	*Inside Macintosh,* Volumes II and V.
ATP	*Inside Macintosh,* Volume II; *Inside Macintosh,* Volume V; and *Inside AppleTalk.*
DDP and Link Access Protocols	*Inside Macintosh,* Volumes II and VI and *Inside AppleTalk.*
NBP	*Inside Macintosh,* Volume II; *Inside Macintosh,* Volume V; and *Inside AppleTalk.*
PAP	*Inside AppleTalk, Inside LaserWriter,* and *Programming the LaserWriter* (David Holzgang, Addison-Wesley, 1991).
RTMP	*Inside AppleTalk.*
ZIP	*Inside Macintosh,* Volume II; *Inside Macintosh,* Volume V; and *Inside AppleTalk.*

Chapter 3 deals with synchronous and asynchronous operations. You will learn about polling, spinning the cursor, using a progress dialog, and other relevant ways to use these two operations.

PART TWO

► A Practical Guide to AppleTalk Programming

Part Two of this book focuses on the practical, nitty-gritty techniques you need to master to write real AppleTalk-based programs. It also discusses each AppleTalk protocol used by application programmers in detail. This part consists of seven chapters.

- Chapter 3 discusses the issues involved in making synchronous and asynchronous calls. When to use each type of call is covered, as well as specific techniques that can be applied when using each type of call.
- Chapter 4 discusses memory management issues you will encounter when programming AppleTalk. It covers parameter block management and other types of data used by AppleTalk.
- Chapter 5 discusses the Name Binding Protocol (NBP). It covers the calls provided for dealing with the NBP. It also covers how NBP works and how you can best make use of it in your own programs.
- Chapter 6 discusses the Zone Information Protocol (ZIP). It covers how to use ZIP in both Phase 1 AppleTalk environments as well as in Phase 2 AppleTalk environments. It also covers important calls provided by AppleTalk to access ZIP.
- Chapter 7 discusses the AppleTalk Transaction Protocol (ATP). It covers two ATP works and how you can best make use of it in your own programs. It also provides a detailed description of how the various ATP calls work.

- Chapter 8 discusses the AppleTalk Data Stream Protocol (ADSP). It covers how to set up and tear down connections and how to transfer data over a connection. It also covers additional features of ADSP such as attention messages and provides a detailed description of the various calls used to access ADSP.
- Chapter 9 discusses a variety of miscellaneous AppleTalk routines and interfaces. It covers opening the various AppleTalk drivers, setting SelfSend mode, getting general AppleTalk information, and using the Chooser to build a user interface.

3 ▶ Synchronous and Asynchronous Operations

When programming AppleTalk, there are two basic ways to interface with it: *synchronously* and *asynchronously*. Synchronous operation means that the program is blocked or prevented from doing any other processing until the requested operation has finished. Asynchronous means that the program can continue with other processing while the asynchronous operation finishes.

▶ Using Synchronous Calls

Synchronous operations are very straightforward to use. A synchronous call simply returns when the operation is complete. There is no fuss or bother with later polling for completion or use of completion routines running at interrupt level.

Of course, synchronous operations have disadvantages too. Because the program is blocked until the call completes, the program can be suspended for long periods of time. This is especially true with AppleTalk, where calls can take many seconds to complete. This behavior is often unacceptable as it locks up the entire machine. Users don't like that.

Another area where synchronous calls cannot be used is while your code is running at interrupt level. Interrupt level is a restrictive environment for your code, and one of those restrictions is that synchronous calls are not allowed.

Yet another concern with using synchronous calls is avoiding race conditions. This is a situation similar to having A wait for B, but B is waiting for A; neither A nor B ever can finish and this hangs the machine. Again, users don't like this.

There are times when one can take advantage of the simplicity of synchronous calls. Simple, fast operations that don't require sending data over the network are fine candidates for use of the synchronous calls. Such things as opening or closing sockets are fine and using the synchronous variation of these calls can simplify your program.

▶ Using Asynchronous Calls

There are two basic variations in using asynchronous calls. The first involves polling for completion of the operation by inspecting the status field in the parameter block. The second involves using completion routines.

▶ Polling for Completion of Asynchronous Calls

Polling for completion of AppleTalk calls works by taking advantage of the fact that the **ioResult** field of the parameter block is set to 1 when the asynchronous call is made. It remains set to 1 while the call is being performed. Once the AppleTalk call is completed, the **ioResult** is changed from 1 to an appropriate result code—either **noErr** or some error condition code.

To use polling with an asynchronous AppleTalk call, you have to inform the AppleTalk call that it should not attempt to call a completion routine when the operation is complete. You do this by setting the **ioCompletion** field of the parameter block to NIL (zero).

This polling technique can be useful in a number of circumstances, for example, spinning the cursor, displaying a progress dialog, yielding to other applications in MultiFinder, or performing parallel operations.

Spinning the Cursor to Show Status

The simplest example of using polling would be to spin the cursor to tell the user that some processing is going on and they should wait for it to finish. To use this technique simply request an asynchronous call, then repeatedly poll the **ioResult** field.

Listing 3-1 illustrates an example of spinning the cursor while you are waiting for an ATP **PSendRequest** call to complete. In line 1, the ATP parameter block is loaded with NIL, signaling that no completion routine will be used.

Listing 3-1. Spinning the cursor

```
 1: anATTPPB.ioCompletion := NIL;
 2: stat := PSendRequest(@anATPPB,kASYNC);
 3: IF stat = noErr THEN BEGIN
 4:    WHILE anATPPB.ioResult = 1 DO BEGIN
 5:       DoSpinMyCursor;
 6:    END;
 7: END;
 8: IF anATPPB.ioResult <> noErr THEN BEGIN
 9:    HandleError(anATPPB.ioResult);
10: END;
```

In line 2, the **PSendRequest** call is issued using the asynchronous version of the call (the constant **kASYNC** would be defined as TRUE). These two actions, the NIL completion routine and the asynchronous version of the call, allow you to use polling to determine when the operation actually does complete.

After checking the status of the **PSendRequest** call in line 3, the **DoSpinCursor** procedure is repeatedly called until the ioResult field of the parameter block is no longer equal to 1.

Errors are checked again in line 8 using the ioResult field of the parameter block. If the **PSendRequest** call completed normally, the ioResult field is set to **noErr**, otherwise it contains the error encountered in the **PSendRequest** call and appropriate action is taken by calling an error handler.

Using a Progress Dialog to Show Activity

A better way to show that the machine is busy (rather than just spinning the cursor) is to present a progress dialog to the user. This allows them to see how long things might take and to see progress being made. This is very important for operations that may take more than a few seconds.

It's also important to provide a cancel button in this dialog so that users can stop the operation if they wish. Often, in network operation, problems can arise and operations take longer than you would expect. When this

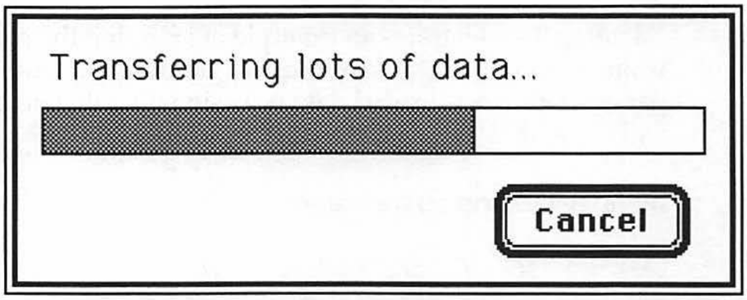


Figure 3-1. A progress dialog

happens, you should give the users the option of stopping the operation and regaining control of their machine.

A useful technique to use in this circumstance is to put up a modal dialog, or better yet, a movable modal dialog, with a process bar and a cancel button as shown in Figure 3-1.

The code to make this work entails repeatedly doing the following:

1. Starting the AppleTalk operation.
2. Calling the **ModalDialog** toolbox trap with a filter routine (this filter routine polls for the completion of the AppleTalk operation and returns a "fake" item press when the ioResult <> 1).
3. Handling the result of the **ModalDialog** call by either continuing, canceling, or responding to an error.

Yielding to Other Applications in MultiFinder

It is also good practice to yield to other programs running under MultiFinder when possible. Quite often AppleTalk calls take a long time to complete; yielding processor resources to the other programs running on the Macintosh makes for a more satisfied user.

For example, say your code is waiting for an ATP request from another program and you're not sure when the request will come in. You can make this request asynchronously and then later you can poll for the completion of the request in your main event loop.

Listings 3-2 and 3-3 illustrate how you would do this. Listing 3-2 shows you how to make the asynchronous request in part of your program while Listing 3-3 is your main event loop.

Listing 3-2. Yielding to another application in MultiFinder—the initial call

```
1: requestATPPB.ioCompletion := NIL;
2: stat := PGetRequest(@requestATPPB,kASYNC);
3: IF stat = noErr THEN BEGIN
4:    pollingState := kWaitingForRequest;
5: END ELSE BEGIN
6:    HandleError(stat);
7: END;
```

This code is similar to the code in Listing 3-1. It shows the parameter block's ioCompletion field being loaded with NIL in line 1 and the call using the asynchronous version of the call (the constant **kASYNC** would be defined as TRUE). Line 4 shows a global state variable being set that will be used later in the event loop (line 2 of Listing 3-3) to keep track of which state you are in.

The actual polling is shown in Listing 3-3. It shows you a normal event loop, with the addition that in line 2 it checks the state variable. If the state is **kWaitingForRequest**, the code then checks the ioResult field of the parameter block. Once this becomes something other than one, processing of the request can take place. Using this model, other parts of your code can perform other asynchronous operations and set the state variable to reflect that. In your main event loop, the case statement would be extended to handle other states. Keep in mind that this example only applies to the case where there is only one asynchronous operation outstanding at any one time. If you want to perform more than one at a time, you need additional state variables.

Listing 3-3. Yielding to another application in MultiFinder—the main event loop

```
1: REPEAT
2:    CASE pollingState OF
3:       kNotPolling: {just continue}
4:       kWaitingForRequest: BEGIN
5:          IF requestATPPB.ioResult = 1 THEN BEGIN
6:             { Do nothing, it's not done… }
7:          END ELSE BEGIN
8:             HandleIncomingRequest;
9:          END;
10:        END;
11:    END;
12:    { Do normal event handling }
13: UNTIL quitingProgram;
```

Parallel Operations

Sometimes while programming AppleTalk it can be helpful to perform more than one operation at the same time. *Parallel operations* (using asynchronous AppleTalk calls in this situation) allow you to do this.

A good example of a parallel operation would be when you are looking up a name in NBP. Typically you only need to look up one NBP type, but it is sometimes the case that you have two NBP types that are important to you. So you would perform two NBP lookup calls. You could, of course, do them one after another, but to save time, it can be faster to issue both calls in quick succession, then poll for their completion. Listing 3-4 shows an example of how this would be done.

Listing 3-4. Performing parallel NBP lookup operations

```
NBPSetEntity(@EntityBuffer1,kNameWildCard,kNBPType1,zoneName);
NBPSetEntity(@EntityBuffer2,kNameWildCard,kNBPType2,zoneName);

{ You fill in the rest of the NBP record information here }

firstMPPPB.ioCompletion  :=  NIL;
secondMPPPB.ioCompletion  :=  NIL;

stat  :=  PLookUpName(@firstMPPPB,kASYNC);
IF stat <> noErr THEN BEGIN
    HandleError(stat);
END;

{ Issue Second NBP Lookup }
stat  :=  PLookUpName(@secondMPPPB,ASYNC);
IF stat <> noErr THEN BEGIN
    HandleError(stat);
END;

{ Now poll for completion }
WHILE  (firstMPPPB.ioResult   = 1)  OR
   (secondMPPPB.ioResult  =  1)  DO  BEGIN
     DoSpinMyCursor;
END;
IF  firstMPPPB.ioResult  <>  noErr  THEN  BEGIN
       HandleError(firstMPPPB.ioResult.ioResult);
END;
IF  SecondMPPPB.ioResult  <>  noErr  THEN  BEGIN
       HandleError(SecondMPPPB.ioResult.ioResult);
END;
```

Lines 1 and 2 set the NBP entities used by each lookup and then fill in the rest of the parameter block on line 4. Lines 6 and 7 get it ready to use a polled asynchronous operation by filling in a NIL completion routine and signaling the asynchronous version of the call (the constant **kASYNC** would be defined as TRUE).

Lines 9 through 18 actually issue the NBP **PLookUpName** calls asynchronously and check for errors. Then, in lines 21 though 30, it waits until both the first and the second call have completed by checking the ioStatus field.

By the Way ▶

Performing AppleTalk operations concurrently can be a powerful tool in creating high performance AppleTalk programs, but there are limitations to this approach. The current implementations of AppleTalk set limits on the number of concurrent operations that can take place on various Macintosh models.

For instance, the example issued concurrent **PLookUpName** calls. When using a Macintosh Plus with the version of AppleTalk found in its ROMs, there is a limit of only one concurrent NBP request at a time. The example code would fail.

On larger Macintoshes and with later versions of AppleTalk, these limits are higher. The ROM-based versions of AppleTalk in the Macintosh SE allow six concurrent NBP lookups; the Macintosh II has a limit of ten. RAM-based versions of AppleTalk raise these limits even further.

These limits are documented in *Inside Macintosh*, Volume V. Your code should be tolerant of these limits and work within the constraints imposed by differences in AppleTalk versions and machine performance.

▶ Using Completion Routines with Asynchronous Calls

When using the preferred AppleTalk interfaces, the asynchronous mode of operation allows a completion routine to be called when a given operation has completed. This is done by setting the asynchronous parameter in the call to TRUE and supplying the address of a completion routine in the **ioCompletion** field of the parameter block.

Completion routines operate in a more limited environment than normal code does. They are called at interrupt level and may not call any trap that can move memory since the heap may not be in a defined state (the Memory Manager may be in the process of moving heap blocks around). A list of the traps that move memory can be found in *Inside Macintosh*, Volumes III, IV, and V.

Completion routines also have another severe constraint imposed upon them: They do not have normal access to data found in other parts of a program. This means that if data needs to be accessed that is not local to the completion routine, some special technique must be used to accomplish this.

The way to deal with this problem is to make use of the fact that completion routines are passed a reference to the parameter block used for that particular operation. To complicate matters further, this reference is passed in processor register A0 and isn't readily accessed via Pascal. A small amount of assembly language is required to get around this.

The technique used throughout this book to access shared data is based on the idea of extending the parameter block referred to by register A0 to include a reference to shared data. The examples always put an additional long word immediately before the regular parameter block. This long word contains an address that points to a block of data that can be used by the completion routine. A small assembly language routine is used to fetch this address and feed it into the Pascal routine as a parameter.

A variation of this approach is to store and restore the program's A5 register so the completion routine can access the program's global variables. This works well when the program is an application. If the program is not an application, storing a reference to A5 clearly doesn't work since A5 is not defined for non-application situations such as resident code installed by an INIT (System 7.0 calls these system extensions).

When using the data block technique, the first step is to identify which types of calls will be used with completion routines. Examine each call and come up with a list of the various parameter block types used. Then, for each of these parameter blocks, define a new record that contains two items: a long word that will be used to store your shared data reference and the real parameter block that will be used by the call. Listing 3-5 shows three examples of extended parameter blocks: one for an ATP parameter block, one for a Time Manager task block, and one for an HFS parameter block.

Listing 3-5. Extended parameter blocks

```
xATPParamBlock = RECORD
          SharedDataRef  : LongInt;
          realATPPB      : ATPParamBlock;
    END;

    xTMTask = RECORD
          SharedDataRef  : LongInt;
          realTMTask     : TMTask;
    END;

    xHParamBlock = RECORD
          SharedDataRef  : LongInt;
          realHPB        : HParamBlockRec;
    END;
```

Once you have defined the extended parameter blocks you require, you need to allocate the shared data record and put its address into the extended parameter blocks before they are used. Listing 3-6 shows an example of this. It allocates a block in the heap using **NewPtr** in line 1. Note that using the **SIZEOF** function on a type returns the size of the type—in this case, the **DataBlockRec**. This allows us to allocate the right amount of storage if we later change the definition of **DataBlockRec**.

Listing 3-6. Setting up an extended parameter block

```
 1: sharedDataPtr := NewPtr(SIZEOF(DataBlockRec));
 2: WITH sharedDataPtr^ DO BEGIN
 3:    { Fill in the data }
 4:    maxTimeout := 5;
 5:    otherData  := 205;
 6: END;
 7: incomingATPPB.SharedDataRef := LongInt(sharedDataPtr);
 8: incomingATPPB.realATPPB.ioCompletion := @preHandleRequest;
 9: stat := PGetRequest(@incomingATPPB.realATPPB,kASYNC);
10: IF stat <> noErr THEN BEGIN
11:    HandleError(stat);
12: END;
```

Lines 3 through 7 would be replaced with your own code that sets initial values of the variables stored in the shared data block. Line 7 is important. It is where the address of the shared data block is stored right before the parameter block using the extended parameter block.

Lines 8 and 9 tell **PGetRequest** to call the **preHandleRequest** completion routine when the asynchronous operation completes.

By the Way ▶

Remember that if you use a local variable for your parameter block, it will be allocated on the stack. This storage is reclaimed when you exit a routine. If the operation does not complete before you exit the current routine, the parameter block will be reused for some other purpose. Chaos will ensue. In this circumstance, you should allocate the parameter block in the heap using **NewPtr**.

Now the tricky part is handling the completion routine itself. When the operation is complete, the AppleTalk drivers call the completion routine at interrupt level, with the address of the parameter block in register A0. The shared data pointer is stored in the four bytes before that, so you access it with a negative offset of four bytes from register A0. Then you move it onto the stack so that the Pascal completion routine can access it as a parameter. Listing 3-7 shows the 68000 assembly language code that does this. As the routine is entered, it stores registers D3 through D7 and registers A2 through A6 onto the stack in line 4. Registers D0 through D2 and registers A0 and A1 are saved by the interrupt handler for you. Later, when the completion routine is finished, these same registers are restored off the stack in line 8.

Listing 3-7. Assembly language completion routine using an extended parameter block

```
1:      IMPORT   HANDLEREQUEST
2: PREHANDLEREQUEST
3:      PROC     EXPORT
4:      MOVEM.L  D3-D7/A2-A6,-(A7) ; save the registers
5:      MOVE.L   -4(A0),D0         ; get our block pointer
6:      MOVE.L   D0,-(A7)          ; pass it as a parameter
7:      JSR      HANDLEREQUEST     ; call the Pascal routine
8:      MOVEM.L  (A7)+,A2-A6/D3-D7 ; restore the registers
9:      RTS
```

Line 5 shows how to get the address of the shared data block from in front of the parameter block. Since register A0 points to the beginning of the parameter block, a negative offset of four from register A0 points to the shared data block address. Line 6 puts that address onto the stack so that the Pascal routine will get it as a parameter. Line 7 actually calls the Pascal routine.

The Pascal completion routine is called with the shared data pointer passed in as a parameter. Once the Pascal completion routine is entered, all the shared data can be accessed by using this parameter. You can explicitly reference it each time using the following syntax: **sharedDataPtr^.yourItem**, where **yourItem** is any field in the shared data record. You must access it this way if you are using C with the syntax: **sharedDataPtr->yourItem**. An alternative approach available with Pascal is to use Pascal's WITH statement; this makes access more convenient.

Listing 3-8 shows how to use an extended parameter block in a Pascal completion routine. Line 1 shows the declaration of the Pascal completion routine with **sharedDataPtr** being passed in as a parameter. The WITH statement on line 3 allows line 4 to refer to any field in the shared data block without prefixing the reference with **sharedDataPtr^**. This approach also leads to more efficient access with most Pascal compilers, since they optimize it and usually place the **sharedDataPtr** into a register.

Listing 3-8. Pascal completion routine using an extended parameter block

```
1: PROCEDURE HandleRequest( sharedDataPtr : DataBlockPtr);
2: BEGIN
3:    WITH sharedDataPtr^ DO BEGIN
4:       { Do the real work of the completion routine }
5:    END;
6: END;
```

▶ Summary

This chapter examined the differences between using AppleTalk protocols synchronously and asynchronously and discussed the circumstances where each approach is appropriate. Examples were given for each type of operation and a number of strategies were covered for using asynchronous calls.

Chapter 4 covers AppleTalk memory management. Using parameter blocks (synchronously and asynchronously) is discussed in detail.

4 ▶ AppleTalk Memory Management

AppleTalk programming on the Macintosh involves using and managing a wide variety of data storage types. Most calls to the AppleTalk drivers involve parameter blocks and many of these calls require additional temporary and permanent storage.

▶ Using Parameter Blocks with Synchronous Calls

Most AppleTalk calls require the use of parameter blocks. These blocks contain the information required by the call as well as the information returned by the call.

When using the synchronous version of the AppleTalk interfaces, parameter blocks are only used briefly. The input data for the call is loaded into the parameter block, the call is made, then the results can be read out of the parameter block. After this, the parameter block is available for reuse in another operation.

The parameter blocks used in synchronous calls can be allocated in a variety of ways. Often the most convenient way is to simply use a local variable. A local variable is allocated on the stack with the helpful side effect of reclaiming the storage once the scope of the local variable is exited.

Listing 4-1 shows an example of this technique. Lines 5–9 show the local variables for the **SendBuffer** routine. The storage for these variables is allocated on the stack as the routine begins execution. Line 23 calls **PSendRequest** synchronously and when the routine exits at line 24, the storage is reclaimed from the stack.

Listing 4-1. Using local storage with synchronous routines

```
PROCEDURE SendBuffer(buffer: Ptr;
                     bufferSize : integer;
                     who : AddrBlock);
VAR
   outPacket    : PacketRec;
   localATPPB   : ATPParamBlock;
   BDSCount     : integer;
   BDSBuff      : str255;
   outBDS       : BDSType;
BEGIN {SendBuffer}
   BDSCount := BuildBDS(@BDSBuff, @outBDS, 255);
   outPacket.size := bufferSize;
   BlockMove(buffer,@outPacket.buffer,bufferSize);
   WITH localATPPB DO BEGIN
      bdsPointer := @outBDS;
      AddrBlock  := who;
      timeOutVal := kStdTimeout;
      retryCount := kStdRetryCount;
      numOfBuffs := 1;
      reqLength  := kMinPacketSize;
      reqPointer := @outPacket;
   END; {WITH}
   CheckStatus(PSendRequest(@localATPPB, kSYNC));
END;
```

Of course there may be times when allocating the parameter blocks in other ways than as local variables may be more appropriate even for synchronous operations. You may wish to allocate some parameter blocks as global variables or use **NewPtr**. This is of course perfectly acceptable.

By the Way ▶

One problem to be aware of when using local variables to hold your parameter blocks is the limited amount of stack space available to your routine. If you attempt to have local variables that take up large amounts of space, be aware of the possibility of overflowing your stack. A particularly dangerous practice is to attempt to declare a large buffer on the stack as a local variable. It only takes one large buffer to blow up your stack.

Another related problem is that when operating at interrupt level you never really know how big your stack is and you should minimize the use of it at all costs. There are some QuickDraw operations that consume large amounts of stack space and if your completion code is called during one of these QuickDraw operations, you may be running with a severely constrained stack. Note that Macintoshes running Color QuickDraw have larger stacks than non-color QuickDraw Macintoshes.

▶ Using Parameter Blocks with Asynchronous Calls

You must be more careful when using parameter blocks with asynchronous calls than you need to be with synchronous calls. Asynchronous calls are not necessarily finished when they return to the caller. Because of this, the parameter block may still be in use long after the call has returned. The parameter block is in use by the asynchronous call until it has completely finished its operation and has signaled this by setting **ioResult** to something other than 1.

Because of the way AppleTalk is implemented on the Macintosh, two important rules must be observed when using parameter blocks with asynchronous calls:

1. A parameter block's integrity must be maintained until the operation is completed.
2. A parameter block cannot be reused until the operation it was used with has completely finished.

▶ Parameter Block Integrity

Maintaining parameter block integrity means that a parameter block does not move in memory and is not written to before the completion of the operation. If you choose to allocate a parameter block from the heap as a handle using **NewHandle,** you must first lock this block before using it in an asynchronous AppleTalk call, and it must remain locked until the operation has completed. AppleTalk expects the parameter block to stay put in memory.

Another aspect of parameter block integrity is that you must not write to a parameter block while it is being used by AppleTalk. A common source of having your parameter block clobbered is to use a parameter block that was a local variable in a subroutine, then exit the routine before the operation has completed. All local variables are allocated on the stack and after exiting the subroutine the stack space is reused.

There are times when using parameter blocks with asynchronous calls that are allocated on the stack is perfectly acceptable (such as while polling for busy cursor feedback). Nevertheless, it is good practice to statically allocate the parameter blocks used with asynchronous calls using **NewPtr** or **NewHandle** with the handle locked when the parameter block is used.

▶ Reuse of Parameter Blocks

The implementation of AppleTalk used on the Macintosh uses the parameter block supplied to a call for the duration of the call. Unlike some other operating systems, no copy is made, but rather the parameter block supplied by the caller becomes the property of AppleTalk until the operation completes. This means that you cannot reuse a parameter block until the call it was used with completes. If you wish to perform parallel operations, multiple parameter blocks are required.

▶ Other Types of AppleTalk Storage

In addition to parameter blocks, AppleTalk uses many other types of permanent data. Often buffers must be allocated for long durations. Care must be taken in these circumstances to make sure that these buffers are left alone while they are in use.

Many examples of these persistent buffers exist throughout AppleTalk. One example is registering a name using NBP. A buffer containing the name, type, and socket for a given name is needed from the time you register the name until the time you unregister it. Another example is the various buffers used by ADSP to hold information relating to a session during the life of that session.

All permanent buffers must be allocated and maintained by your program for the duration of their use. Allocating them as local variables must be done with care to assure that the buffers are not lost when exiting the local scope. Most buffers are appropriately managed as global variables or they are allocated from the heap.

▶ Additional AppleTalk Storage Management Considerations

There are other essential considerations when dealing with Macintosh memory management. If you are frequently allocating and deallocating buffers using **NewHandle** or **NewPtr** and the heap is very fragmented, the Memory Manager may have to move large amounts of data around to satisfy your request. This can introduce noticeable pauses into your program.

Another area to take extra care over is your use of system permanent memory, such as memory in the system heap, or the memory above **BufPtr**. This memory is permanently taken away from the user and there is no way for the user to get it back. Often there is no alternative to using this memory, but always minimize its use. Note that Apple recommends against using the memory above **BufPtr** with System 7.0 and later systems.

▶ Avoiding Heap Fragmentation

Memory management is always important when programming the Macintosh. Using AppleTalk requires additional considerations when you are planning your application's memory management strategy. A brief discussion of these considerations follows.

The memory buffers AppleTalk requires are commonly allocated with a call to **NewPtr**. This returns a block of memory of the desired size referenced by a pointer. Because these blocks allocated by **NewPtr** are allocated in the heap as non-relocatable blocks, care must be taken not to fragment the heap.

Heap fragmentation comes from allocating non-relocatable blocks in the middle of your heap. The Memory Manager attempts to put new non-relocatable blocks as low in the heap as possible to lessen the chances of heap fragmentation. Nevertheless, these blocks can sometimes be allocated in the middle of the heap when your heap becomes full.

If your program is using a fixed number of parameter blocks and buffers, often the best way to allocate them is to do so during the initialization of your program before other large objects are allocated on your heap. Another approach is to simply use global variables.

By the Way ▶

A common problem occurs when debugging AppleTalk code that allocates buffers for AppleTalk. When your code crashes and you do an Exit-to-shell command from your debugger to get back to MultiFinder, the buffers owned by AppleTalk are still in use. Later when you run another program in the same memory space as your old, dead program, these buffers get clobbered and this can lead to system bombs.

Consider the example of registering an NBP name. When this is done, NBP stores the name, type, and socket of the entity into the buffer supplied by the program. If this program crashes, the buffer is still used by NBP and when that memory is later used by another program, the buffer gets clobbered and NBP gets very confused.

There is a solution for this. Patch the **ExitToShell** trap so that your patch code can do basic housecleaning at that time. It can then unregister NBP names, cancel outstanding I/O calls, and so forth.

▶ Non-Application Storage Management

Code that does not execute as a standard application has special needs when it comes to storage management. If your code is installed at startup time by an INIT, it obviously can't make use of an application heap for its data storage needs. There are basically two places available for data storage for resident code installed at startup time: the system heap and the memory above **BufPtr**.

BufPtr is a low-memory global that points to the logical top of memory. The space between where **BufPtr** points and the physical end of memory can be used to store static data. This is done by moving **BufPtr** down by the amount of storage that you require. For example, if your INIT needs 32K for some purpose and you wish to store it above **BufPtr**, you would move **BufPtr** down by 32K by subtracting 32768 from **BufPtr**. Listing 4-2 illustrates how you might do this.

Listing 4-2. Allocating permanent storage using BufPtr

```
1: TYPE SomeBufferType = ARRAY[0..32767] of Byte;
2:
3: Handle(BufPtr)^ := Handle(BugPtr)^ - SIZEOF(SomeBufferType);
4: aBufferPtr := Handle(BufPtr);
```

Line 1 defines a 32K buffer type.

Line 3 moves **BufPtr** down by the size of the buffer type.

Line 4 saves the address of the space that was reserved above **BufPtr.**

The other approach for using permanent data storage is to use the system heap. The system heap is always available and is never destroyed (unlike the various application heaps) so data placed in it can be accessed anytime.

To assist in the use of system heap space, the Startup Manager provides a mechanism for INITs (and RDEVs too) to request a certain amount of system heap space when they are run. This mechanism relies on the use of a 'sysz' resource with ID=0 located in the INIT file. If this 'sysz' resource exists in your INIT file, the system will look at the first long word of data contained in the resource and try to make sure that there is this much space available in the system heap for your use. Of course, there are limits to the amount of space available on any machine, but the Startup Manager does guarantee that there will be at least 16K of contiguous space available to you.

To make use of the system heap space, you have to set the current zone to be the system zone using the Memory Manager **SetZone** trap; use the function **SystemZone** to return the system zone pointer. Then use normal Memory Manager traps such as **NewPtr** or **NewHandle** to allocate blocks from that heap.

Resource Manager calls can also be used to load resources into the system heap. Using the system heap flag for the resource ensures that the resource is loaded into the system heap. Listing 4-3 illustrates how this can be done. It starts at line 1 by setting the current heap zone to be the system heap. Then it allocates a block in the system heap of the appropriate size on line 2.

Listing 4-3. Allocating permanent storage from the system heap

```
 1:  SetZone(SystemHeap);
 2:
 3:  dataBlockPtr  :=DataBlockPtr(NewPtr(SIZEOF(DataBlockRec)));
 4:  FailNil(dataBlockPtr);
 5:
 6:  codeH  :=  Get1Resource('CBlk',kMyResidentCodeBlockID);
 7:  FailNil(codeH);
 8:  DetachResource(codeH);
 9:  HNoPurge(codeH);
10:  HLock(codeH);
11:
12:  SetZone(ApplicZone);
```

Line 6 brings the code resource into the system heap and lines 8, 9, and 10 make sure that it is detached from the resource map, locked, and not purgeable. Note: If you don't detach the resource from the resource map, it will be purged when the resource file is closed.

At times you may want to alert the user about something important that happens when your application is running in the background. To do so, you can use the Notification Manager to install a notification request. You request a notification by passing the Notification Manager the address of a notification record, which contains information about the ways in which the Notification Manager should alert the user. **nmRefCon**, one of the fields in the notification record, is a reference constant that is reserved for your application's use. When you set up a notification request, you can use that field to hold a reference to your own storage.

▶ Summary

This chapter discussed the issue of memory management as it pertains to using the AppleTalk protocols. Allocating parameter blocks and other types of AppleTalk-related storage was covered. Special attention was given to storing data outside the bounds of a running application using the system heap or the memory above **BufPtr**.

Chapter 5 discusses Name Binding Protocol. It covers how to handle network names and offers detailed descriptions of specific NBP routines.

5 ► Name Binding Protocol

Name Binding Protocol (NBP) is a basic component of AppleTalk. It provides a way for programs to find each other over the network.

A good analogy for NBP is directory assistance in the telephone system. Directory assistance allows you to get the telephone number of someone else by making a special call to the directory assistance operator and asking for them by name. With that information, the operator can then tell you the specific phone number you want. NBP works much like this: You can ask for a specific name on the network, and get a network address back.

NBP provides services for registering a name for an address, looking up an address given a name, confirming a name, and unregistering a name.

► Network Addresses

AppleTalk has a very specific way of identifying network entities. It's called the entity's *address* and is made up of three components: the socket number, the node ID, and the network number.

The first part of the entity's address, the *socket number*, identifies the specific socket on a given node. Each entity can have one or more unique sockets assigned to it.

The second part of the entity's address, the *node ID*, identifies the computer the program is running on (a Macintosh, LaserWriter, or other network computer). The node ID is unique on a network and is set at the time the computer is turned on and connected to the network. It can change after a node is either reset or turned off and back on.

By the Way ▶

AppleTalk uses a rather clever scheme for selecting node IDs. When a Macintosh is turned on, it selects a node ID. (Macintoshes remember their last ID in PRAM, but other AppleTalk devices may not remember their ID when powered off and randomly select a new one each time they are turned on.) It then attempts to communicate with another node on the network with the same ID. If it gets a reply, meaning some other node already has that ID, it picks another node ID and repeats the process until it finds a node ID that no other node is using.

One benefit of this scheme is that nodes can be moved onto other networks with no special configuration process. For example, if I have a Mac that I use on one network and it last operated as, say, node 15, I can take that Mac to another network that already has a device that thinks it is node 15. When I connect my Mac to that network, AppleTalk will see that another computer has already been assigned node 15 and try another node number, say 16, and repeat this process until it can find a node that is not in use.

With this simple scheme, AppleTalk avoids many of the configuration headaches that plague some other network protocols.

The third part of the entity's address, the network number, identifies the *network* the Macintosh is connected to. Networks are a number of computers connected by a single logical wire, for example a daisy chain of LocalTalk cables or a single Ethernet cable. Groups of networks are connected by routers and these groups of networks form internets. The internet routers maintain information about the other networks in an internet and pass data between networks so that the data will ultimately arrive at the right place.

Note ▶

Small AppleTalk networks do not require routers because the local network can always be referred to as network number zero. This allows any address containing network zero to be delivered without going through a router. Even in large internets, network number zero can still be used to refer to an address on the local network without explicitly using the network's number.

▶ NBP Names

NBP names are made up of three components: object, type, and zone. These three components are often written in sequence with a colon (:) separating the object and type, and an at-sign (@) separating the type from the zone. Here's how the components look when written out:

object:type@zone

Each component is a string of up to 31 characters.

The *object* part of the name typically identifies the Macintosh where the name resides. It is almost always set to the Chooser name that the user has selected. Sometimes, an additional identifier is appended to the Chooser name when a duplicate name is found, for example, "Jose Smith" and "Jose Smith 2."

The Chooser name is stored as a resource of type 'STR' with ID = -16096 in the System file. Because the resource fork of the System file is always available to applications (and INITs too), your program can retrieve this name by using a call to the **GetString** trap for the string with the ID of -16096.

By the Way ▶

In System 7, the so-called Chooser name is no longer set in the Chooser. Instead, there is a Control Panel called Network Setup that handles this function. This Control Panel lets the user assign two names: the Owner name and the Macintosh name. The Owner name is used to identify the user of a given Macintosh and is stored in 'STR' ID = -16096. This is the same as the old Chooser name. The Macintosh name is used to identify a particular Macintosh and is stored in 'STR' ID = -16413.

For example, if a user had two machines, he or she could have the same Owner name, say "Pat," on each machine, but then different Macintosh names, say "Pat's Classic" and "Pat's Mac IIsi."

With this new naming scheme, it is often more appropriate to use the Macintosh name to identify services on a Macintosh. Use the Owner name when specifically referring to the user rather than the machine.

The *type* part of the name is used to differentiate between different types of services on a given Macintosh. This should be a unique string that identifies the service. For example, "LaserWriter," "AFPServer," or "Public Folder," are used to identify LaserWriters, AppleShare servers, and Public Folders.

The *zone* part of the name identifies which zone the name resides on. Zones are used to identify which network a given name belongs to. Zones are managed by internet routers using the Zone Information Protocol (ZIP).

▶ Wildcards in NBP Names

NBP allows you to make use of certain special characters to match more than one name at a time. These special characters are called *wildcards*.

Both object and type strings use the equal (=) wildcard to match all possible values. For example, the "=:LaserWriter@HQEthernet" string would match all LaserWriters in the HQEthernet zone. Likewise, the "Bob:=@HQEthernet" string would match all types of entities with the name Bob in the HQEthernet zone.

In the zone string, the asterisk (*) wildcard is used to signify the default value. This is either the zone that the node is currently in, or no zone when there is only a simple AppleTalk network setup and zones are not needed to differentiate between multiple networks.

By the Way ▶

When using the newer AppleTalk Phase 2, the double tilde (≈) can be used as an additional wildcard. It matches zero or more characters when used in the object or type string. It has no special meaning in the zone string. Only one double tilde is allowed per string. This allows more elaborate matching to be performed. For example, you could look up the "Mary≈:PublicFolder@HQEthernet" string and it would match both Mary Smith's and Mary Jones's Public Folder in the HQEthernet zone.

▶ Using Name Binding Protocol

There are two basic network functions that you can perform with NBP: registering your name so that others can see it, and looking up someone else's name. In many situations you will do both. This section explains these two options as well as how to confirm a name on the network.

▶ Registering a Name on the Network

To register a name for a given socket, you use the **PRegisterName** trap. You will also need to create an NBP names table element for it. Use the **NBPSetNTE** trap to supply **PRegisterName** with the desired name, type, and zone strings. Listing 5-1 shows how this is done.

Listing 5-1. Registering an NBP name on the network

```
NBPSetNTE(@theElementBuff,theName,theType,'*',theSocket);
WITH theMPPPB DO BEGIN
   entityPtr  := @theElementBuffer;
   interval   := 3;
   count      := 2;
   verifyFlag := $FF;
 END;
CheckStatus(PRegisterName(@theMPPPB,kSYNC));
```

Line 1 is the call to **NBPSetNTE**. The first parameter is a pointer to a buffer that will contain the names table element. The next three parameters are the NBP name, type, and zone strings. And the final parameter is the socket that is being assigned the name.

Lines 2–7 set up the parameter block with the entity buffer, retry information (it retries twice, waiting 24 ticks between each retry), and set the **verifyFlag** to TRUE ($FF) so it will check for duplicate names.

Line 8 actually issues the **PRegisterName** trap using the synchronous variation. It also calls a status-checking routine to handle any errors that could be returned from the trap.

▶ Looking Up a Name on the Network

You use the **PLookupName** trap to look up a name on the network. You give it an entity name, and it returns a list of matching names and addresses in a buffer. You then extract the names and addresses from the buffer using the **NBPExtract** trap.

Important ▶ **PLookupName works for looking up fully qualified names as well as those with wildcards. Be careful when looking up names tagged with wildcards to pass in a large enough buffer; the buffer must be large enough to hold all the names that match.**

To use the **PLookupName** trap, you need to create an NBP entity name to supply to the **PRegisterName** trap. This is done by using the **NBPSetEntity** trap, supplying it with the desired name, type, and zone strings. **NBPExtract** is also used after **PLookupName** to extract the address and actual entity name from the buffer returned from **PLookupName**.

Listing 5-2 shows how you would look up a specific name and get its address.

Listing 5-2. Looking up an NBP name on the network

```
 1: NBPSetEntity(@theEntityBuffer,theName,theType,theZone);
 2: WITH theMPPPB DO BEGIN
 3:    entityPtr  := @theEntityBuffer;
 4:    retBuffSize:= kReturnBufferSize;
 5:    retBuffPtr := @theReturnBuffer;
 6:    maxToGet   := 1;
 7:    interval   := 3;
 8:    count      := 2;
 9:    verifyFlag := $FF;
10: END;
11: CheckStatus(PLookupName(@theMPPPB,kSYNC));
12: IF theMPPPB.numGotten > 1
13:    THEN BEGIN
14:       CheckStat(NBPExtract(@theReturnBuffer,
                 theMPPPB.numGotten,1,theEntity,theAddress));
15:    END;
```

Line 1 is the call to **NBPSetEntity**. It converts the name, type, and zone strings provided into an NBP entity stored in **theEntityBuffer**.

Lines 4 and 5 set the return buffer information. The return buffer should be large enough to contain the number of names tuples expected to match the specified entity. If it's too small, the error **nbpBuffOvr** will be returned.

Line 6 tells it to only look for one match.

Lines 7 and 8 tell it to retry twice, waiting 24 ticks between each retry.

Once the call is made, and if there are no errors, **NBPExtract** is used in Line 14 to get the address for the given entity.

▶ Confirming a Name on the Network

To confirm a name on the network, use the **PConfirmName** trap. You give it an entity name and address and it tells you if that name and address are still associated with each other.

PConfirmName should be used when some time has passed since you originally looked up a name. It is much more efficient than using **PLookupName** because it can ask the specified machine directly rather than doing a general lookup.

To use this trap you need to create an NBP entity name to supply to the **PRegisterName** trap. Use the **NBPSetEntity** trap, supplying it with the desired name, type, and zone strings.

Listing 5-3 shows how you would confirm an NBP name and get its new socket address if it has changed.

Listing 5-3. Confirming an NBP name on the network

```
 1: NBPSetEntity(@theEntityBuffer,theName,theType,theZone);
 2: WITH theMPPPB DO BEGIN
 3:    entityPtr  := @theEntityBuffer;
 4:    interval   := 3;
 5:    count      := 2;
 6:    confirmAddr:= oldAddress;
 7: END;
 8: status := PConfirmName(@theMPPPB,kSYNC);
 9: CASE status OF
10:    noErr       : {The name & address were confirmed};
11:    nbpNoConfirm: {The name was not confirmed}
12:    nbpConfDiff : {The name was assigned another socket};
13:         oldAddress.socket := theMPPPB.newSocket;
14: END;
```

Line 1 is the call to **NBPSetEntity**. It converts the name, type, and zone strings provided into an NBP entity stored in **theEntityBuffer**.

Lines 4 and 5 tell it to retry twice, waiting 24 ticks between each retry.

Line 6 tells it which address to confirm.

Line 8 makes the call to **PConfirmName** synchronously.

Lines 10–12 handle the three possible return statuses: confirmation, no confirmation, and confirmation at another socket. If the socket has changed, Line 13 shows how to update the address by using the **newSocket** field to update the socket portion of the old address.

▶ Detailed Descriptions of Important NBP Routines

The following section describes each important NBP routine. Each routine's prototype is shown, and all parameters or parameter block fields and error codes are listed. Each parameter or parameter block field and error code is then described in detail.

▶ PRegisterName

PRegisterName is used to get your NBP name registered on the network. All code that creates an NBP name for others to see will use the following routine.

```
FUNCTION PRegisterName(thePBPtr:MPPPBPtr; async:
BOOLEAN):OSErr;

Using the following fields in the MPP Parameter Block:

  → ioCompletion       - address of completion routine
  ← ioResult           - result of operation
  → interval           - retry interval in eight tick units
  ↔ count              - retry count
  → entityPtr          - names table entry pointer
  → verifyFlag         - should verification be done

Errors Returned:

    noErr (0)                No error.
    nbpDuplicate (-1027)     This name already exists.
    nbpNISErr (-1029)        Error opening the NIS.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

interval and **count** determine the retry behavior of the trap. **count** tells it how many retries should be attempted and **interval** tells it how long to wait between retries in 8–tick units. **count** is decremented with each retry.

entityPtr contains a pointer to an NBP names table entry. This is usually built with the **NBPSetNTE** trap. It contains a names table entry plus some additional information stored by NBP; it is 108 bytes in size.

Note ▶

No wildcards are allowed in the object name and type portions of the name and the zone portion must be set to an asterisk (*).

entityPtr does not point to a **NameEntity** in this case, but rather a names table entry. The names table entry is owned by NBP until it is released using the **RemoveName** trap. Make sure that this memory is not reused in your program!

verifyFlag should always be set to TRUE. There are some rare circumstances when this flag can be set to FALSE , but this is almost never done in application code. When set to TRUE, **verifyFlag** tells NBP to look for the given name on the network to check for duplicates.
noErr is returned when the trap completes normally.
nbpDuplicate is returned when the specified name is already in use.
nbpNISErr is returned when AppleTalk is unable to open the Names Information Socket.

▶ PLookupName

PLookupName is used to get the address of an NBP name. It also handles looking up a set of names using the NBP wildcards and will return a list of the matching names and their corresponding addresses. The **PLookup-Name** routine follows.

```
FUNCTION  PLookupName(thePBPtr:MPPPBPtr;  async:  BOOLEAN):OSErr;

Using  the  following  fields  in  the  MPP  Parameter  Block:

  →  ioCompletion        -  address  of  completion  routine
  ←  ioResult            -  result  of  operation
  →  interval            -  retry  interval  in  eight  tick  units
  ↔  count               -  retry  count
  →  entityPtr           -  names  table  entry  pointer
  →  retBuffPtr          -  pointer  to  match  data  buffer
  →  retBuffSize         -  match  data  buffer  size
  →  maxToGet            -  how  many  matches  to  try  to  get
  ←  numGotten           -  how  many  matches  were  found

Errors  Returned:

  noErr  (0)               No  error.
  nbpBuffOvr  (-1024)      Error  opening  the  NIS.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.
ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.
interval and **count** determine the retry behavior of the trap. **count** tells it how many retries should be attempted and **interval** tells it how long to wait between retries in 8-tick units. **count** is decremented with each retry.

entityPtr contains a pointer to an NBP entity. The NBP entity is usually built with the **NBPSetEntity** trap. An NBP entity is 99 bytes in size. Wildcards are allowed in this entity.

retBuffPtr and **retBuffSize** describe a buffer used to hold the result of the lookup. If the return buffer is too small to hold all the names that match, **nbpBuffOvr** will be returned.

maxToGet is the maximum number of names that the trap should try to find. If this number of names is found before all the retries are done, the trap will complete early.

numGotten is the number of names that were actually found on the network that matched the specified entity.

noErr is returned when the trap completes normally.

nbpBuffOvr is returned when there are more names matching the specified entity than there is buffer space to hold the result.

▶ NBPSetEntity

NBPSetEntity is a utility routine provided to assist you in creating NBP entity data structures. It takes three strings—the object name, type, and zone—and puts them into an entity buffer. Here's how the **NBPSetEntity** utility routine is written:

```
PROCEDURE  NBPSetEntity(buffer  :  Ptr;  nbpObject,  nbpType,nbpZone :
Str32);
```

buffer is a pointer to where the new entity will be stored. It must be at least 99 bytes long.

nbpObject, nbpType, and **nbpZone** are the three strings that make up the name of the new entity. Each of these strings can be up to 31 characters in length.

▶ NBPSetNTE

NBPSetNTE is a utility routine provided to assist you in creating names table entries. It takes three strings—the object name, type, and zone—as well as a socket number and puts them into a names table entry. The **NBPSetNTE** routine follows.

```
PROCEDURE  NBPSetNTE(ntePtr   :  Ptr;  nbpObject,  nbpType,nbpZone :
Str32;  socket  :  integer);
```

ntePtr is a pointer to where the new names table entry will be stored. It must be at least 109 bytes long.

nbpObject, **nbpType**, and **nbpZone** are the three strings that make up the name of the new entity. Each of these strings can be up to 31 characters in length.

socket is the number of the socket that will be associated with the entity specified.

▶ NBPExtract

NBPExtract is a utility routine provided to assist you in extracting information from the list of names and addresses returned by **PLookupName**. It takes a pointer to the buffer and an index into the table and returns the specified entry's name and address. This is how the **NBPExtract** routine looks:

```
FUNCTION NBPExtract(theBuffer : Ptr; numInBuf : INTEGER; whichOne:
INTEGER; VAR abEntity : EntityName; VAR address : AddrBlock) : OSErr;
```

theBuffer is a buffer pointer containing raw entity name and address information. This is normally the return buffer from **PLookupName**.

numInBuf is the number of entity names and addresses that can be found in the buffer. This is normally the **numGotten** field from **PLookupName.**

whichOne tells the trap which one of the various entity names and addresses should be retrieved from the buffer.

abEntity is the entity name returned by the call.

address is the address returned by the call.

noErr is returned when the trap completes normally.

▶ NBPRemove

Using **NBPRemove** removes a previously registered NBP name. Use it when you are finished with a given NBP name and no longer wish others to be able to see it on the network. The **NBPRemove** routine follows.

```
FUNCTION NBPRemove(abEntity: EntityPtr): OSErr;
```

abEntity is the entity name that you want to remove from the local names table.

noErr is returned when the trap completes normally.

nbpNotFound is returned when the specified entity name is not found in the local names table.

▶ PConfirm

PConfirm confirms that a specified name is associated with a particular address. If the address is no longer associated with the name, a number of possible circumstances may have occurred, such as the name being removed from the target socket, or the target node could have been rebooted with the name being assigned to a different socket. **PConfirm** should be used instead of **PLookupName** because it uses fewer network resources to accomplish its task. The **PConfirm** routine follows.

```
FUNCTION PConfirm(thePBPtr: MPPPBPtr; async: BOOLEAN)): OSErr;

Using the following fields in the MPP Parameter Block:

  → ioCompletion        - address of completion routine
  ← ioResult            - result of operation
  → interval            - retry interval in eight tick units
  ↔ count               - retry count
  → entityPtr           - names table entry pointer
  → confirmAddr         - address to confirm
  → newSocket           - socket number to confirm

Errors Returned:

  noErr (0)             No error.
  nbpConfDiff (-1026)   Name confirmed at different socket.
  nbpNoConfirm (-1025)  Name not confirmed.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation, this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

interval and **count** determine the retry behavior of the trap. **count** tells it how many retries should be attempted and **interval** tells it how long to wait between retries in 8-tick units. **count** is decremented with each retry.

entityPtr contains a pointer to the NBP entity you want to confirm.

confirmAddr is the address you want to confirm.

newSocket returns the socket number of the entity you confirmed. This will be different from the socket found in **confirmAddr** when the status **nbpConfDiff** is returned.

noErr is returned when the trap completes normally.

nbpConfDiff is returned when the entity you just checked isn't at the same socket specified in **confirmAddr**. The new socket number can be found in **newSocket** when this status is returned.

nbpNoConfirm is returned when NBP cannot confirm that the specified entity name is associated with the specified address.

▶ Summary

This chapter described the operation of the Name Binding Protocol. After discussing what names and address are, it detailed how to make use of them in your own programs by providing examples of code fragments. Furthermore, each NBP routine that you need to use in order to incorporate network names into your programs was described. Each parameter or parameter block field used in each routine was discussed in detail.

Chapter 6 discusses Zone Information Protocol for Phase 1 and Phase 2 AppleTalk networks. These discussions are followed by descriptions of specific ZIP routines.

6 ▶ Zone Information Protocol

The Zone Information Protocol (ZIP) provides three basic functions that are useful to most application programs:

- getting the list of all zones on a network
- getting the list of all local zones (available only on Phase 2 AppleTalk networks)
- getting the name of the current zone

ZIP provides a fairly low-level interface. For its most basic functions (normally not required by application programs), ZIP requires that you use Datagram Delivery Protocol (DDP) to send and receive information. For the information needed by most application programs, ZIP provides an ATP-based interface. Although easier than handling DDP packets, this does require building ATP requests in a certain format, setting the proper user bytes, and repeating transactions to get all the information you need. The requirement that you use ATP has been removed in Phase 2 AppleTalk with the addition of ZIP specific calls to the .XPP driver, although the older technique is still supported.

This chapter shows how to use ZIP with both Phase 1 and Phase 2 AppleTalk networks. If you are confident that your code will not be operating in Phase 1 environments, you can ignore the discussion of ZIP in Phase 1 AppleTalk networks.

▶ Using ZIP in Phase 1 AppleTalk Networks

Using ZIP in Phase 1 AppleTalk requires you to build requests and get responses using ATP in order to retrieve your current zone name or to retrieve a list of all available zones. Each type of request is specified by a unique value stored in the **userBytes** field of the ATP message.

▶ Getting the Local Zone Name

To get the name of the current zone from ZIP in Phase 1 AppleTalk, you have to build a **GetMyZone** request using ATP. This ATP request doesn't have an ATP message body; it only specifies the ZIP function in the user bytes. This is done by setting the first byte of the ATP user bytes to be 7 because **GetMyZone** is ZIP function 7. The other user bytes should be set to zero.

The **GetM4yZone** message should be sent to your local router. Its address, made up of the socket number, the node ID, and the network number, are generated in the following way.

1. The router's ZIP information socket number is always 6. Unlike most socket addresses used by AppleTalk, the ZIP information socket in a router is always statically assigned to be socket number 6.
2. The node ID of your local router is retrieved by making a call to the **GetBridgeAddress** call. If this returns zero, it means that there is no router available and that the current network is made up of only one zone.
3. The network number of the router is the same as your local network number. This can be retrieved by making a call to **GetNodeAddress.**

Listing 6-1 shows a function that implements these three steps.

Listing 6-1. GetZIPAddr function

```
FUNCTION GetRouterAddr(VAR ZIPAddr : AddrBlock) : BOOLEAN;
CONST
    kZIPSocket = 6;
VAR
    theNode       : integer;
    theNet        : integer;
    theRouterNode : integer;
BEGIN {GetZIPAddr}
    GetRouterAddr := FALSE;
```

```
10:     IF GetNodeAddress(theNode, theNet) <> noErr
11:         THEN BEGIN
12:             EXIT(GetZIPAddr);
13:         END;
14:     theRouterNode := GetBridgeAddress;
15:     IF (theRouterNode = 0)
16:         THEN BEGIN {No zones}
17:             EXIT(GetZIPAddr);
18:         END;
19:     WITH ZIPAddr DO BEGIN
20:         aNet    := theNet;
21:         aNode   := theRouterNode;
22:         aSocket := kZIPSocket;
23:     END;
24:     GetRouterAddr := TRUE;
25: END; {GetZIPAddr}
```

Line 10 makes the call to **GetNodeAddress**, exiting the function with a value of FALSE if there is any error encountered.

Line 14 calls **GetBridgeAddress** to fetch the node ID of the router. Line 15 checks for a result of zero indicating that there is no router on the network.

Lines 19–23 assign the address values to the return parameter **ZIPAddr**. Then line 24 sets the return value of the function to be TRUE, indicating that the function succeeded in getting the address of the router for use with ZIP.

▶ Getting the Zone List

In order to get the list of all the defined zones in a network from ZIP in Phase 1 AppleTalk, you have to build a **GetZoneList** request using ATP. This ATP request contains no ATP message body, but only specifies the ZIP function in the user bytes. This is done by setting the first byte of the ATP user bytes to be 8 because **GetZoneList** is ZIP function 8.

In addition to setting the first user byte to 8, the **GetZoneList** message requires you to supply an index in the last two bytes of the user bytes. This index mechanism is needed because often a complete zone list cannot fit into a single response packet. In this case, subsequent requests are made for additional parts of the list until the entire list is completely retrieved.

The first time you make the **GetZoneList** request, you should set the index to 1. This tells the router to give you the first portion of the zone list. In the response packet, the router indicates how many zones are listed in the packet and if there are any more zones left to be received.

The last two user bytes are used to contain the zone count of the packet; the first byte contains the flag indicating if there are additional zones to be gotten.

Listing 6-2 shows how to define a Pascal type that directly maps the response user byte into a more tractable form. The first field, **IsLastPacket**, corresponds to the first byte of the user bytes and tells you if the packet is the last one needed to fetch an entire list. A filler field is supplied to skip over the second user byte that is unused. Then, the **ZoneCount** field is mapped over the final two user bytes that tell you how many zones are in the packet.

Listing 6-2. Alternate type for the GetZoneList user bytes

```
GetZoneUserBytes = PACKED RECORD
  IsLastPacket: BOOLEAN;
  Filler      : 0..255;
  ZoneCount   : integer;
END;
```

Listing 6-3 is a code fragment that shows how to repeatedly send the **GetZoneList** ATP request and process the response. In this example, a call is made to **ProcessZone** for each zone name that is received.

Listing 6-3. GetZoneList loop in Phase 1 AppleTalk

```
index := 1;
count := 0;

REPEAT
  theATPPB.userData := kGZLCall + index;
  stat := PSENDRequest(@theATPPB, kSYNC);
  IF stat <> noErr
    THEN handleError(stat);
  count := count +
     GetZoneUserBytes(theBDS.userBytes).ZoneCount;
  currZonePtr := zonePtr;
  REPEAT
    zoneName := StrFromPtr(currZonePtr,currZonePtr^);
    ProcessZone(zoneName);
```

```
15:      currZonePtr := Ptr(LONGINT(currZonePtr) +
16:                              currZonePtr^+1);
17:      index := index + 1;
18:    UNTIL index > count;
19:  UNTIL (GetZoneUserBytes(theBDS.userBytes).IsLastPacket);
```

Lines 1 and 2 initialize the counters that keep track of how many zone list names you have processed.

Line 4 begins the loop for sending ATP requests. Each time through the loop, you need to send the proper index and function code. This is set in line 5 by adding the index to the function, with the function already shifted over into the first user byte position (kGZLCall = $08000000).

Line 6 makes ATP send the request synchronously and lines 7 and 8 check for any errors, calling an error handling routine if there is one.

Lines 9 and 10 increment the count variable by the number of zones returned in the response packet using the **GetZoneUserBytes** structure that was defined in Listing 6-2.

Lines 12–18 define a loop that repeats for each zone listed in the response packet.

Line 13 uses a routine, **StrFromPtr**, that copies a string starting at the specified address for the specified length. **StrFromPtr** copies the zone name into the **zoneName** variable.

Line 14 feeds the zone name from line 13 into a function that can then process the zone name as it sees fit.

Lines 15 and 16 move the **currZonePtr** to the next zone name.

Line 17 bumps up the index and line 18 checks it to see if it is done processing the response packet.

Finally, line 19 checks to see if the packet is the last packet in the zone list. It does this by checking the flag in the user bytes using the **GetZoneUserBytes** structure defined in Listing 6-2.

▶ Using ZIP in Phase 2 AppleTalk Networks

Instead of constructing the ATP messages and sending them yourself in order to use ZIP in Phase 2 AppleTalk, the .XPP driver now includes calls to do this for you. You still have to make repeated calls to the .XPP driver because each call will only get a single buffer full of the zone name data. However, this simplifies the use of ZIP substantially.

In order to make the ZIP calls to the .XPP driver, you need to make a **PBControl** call to the .XPP driver with the **csCode** field in the parameter block set to 246, meaning this is an .XPP driver call. The **xppSubCode** field must be set to either 7 for **GetMyZone**, 6 for **GetZoneList**, or 5 for **GetLocalZones**. You also need to have the **ioRefNum** set to the correct value for the .XPP driver. The easiest way to get this is to simply open the .XPP driver and use the reference number that the open call returns.

Another enhancement found in Phase 2 ZIP is the ability to get a list of all the local zones for your network. This is important because with Phase 2 AppleTalk, a given local network can contain more than one zone. This is often found in larger Ethernet-based AppleTalk networks.

▶ Getting the Local Zone Name

To get the local zone name under AppleTalk Phase 2, you need to make a call to **PBControl** with the parameter block's **csCode** field set to 246 and the **xppSubCode** field set to 7.

Listing 6-4 shows a function that gets the local zone using the **PBControl** call to the .XPP driver.

Listing 6-4. Getting the local zone

```
FUNCTION GetMyZonePhase2 : str255;
CONST
    kXPPCall    = 246;
    kGetMyZone  = 7;
VAR
    theXPBPB        : xCallParam;
    xppDriverRefNum : integer;
    returnedZoneName: str255;
BEGIN {GetMyZonePhase2}

    GetMyZonePhase2 := '*';

    IF OpenDriver('.XPP', xppDriverRefNum) <>  noErr
        THEN EXIT(GetMyZonePhase2);

    WITH theXPBPB DO BEGIN
        zipInfoField[1] := 0;
        zipInfoField[2] := 0;

        ioRefNum    := xppDriverRefNum;
        csCode      := kXPPCall;
        xppSubCode  := kGetMyZone;
```

```
23:          xppTimeOut   := kATPTimeOutVal;
24:          xppRetry     := kATPRetryCount;
25:          zipBuffPtr   := @returnedZoneName;
26:       END;
27:
28:       IF PBControl(@theXPBPB,kSYNC) = noErr
29:          THEN BEGIN
30:             GetMyZonePhase2 := returnedZoneName;
31:          END;
32:
33:    END; {GetMyZonePhase2}
```

Line 11 sets the return value to '*' by default. This will be returned if there are no zones found or there is some other error condition encountered inside the function.

Lines 13 and 14 open the .XPP driver. This is a good way to get the reference number for the .XPP driver. If any error is encountered, the code exits the function without doing any further processing.

Lines 17 and 18 set the first word in the **zipInfoField** field of the parameter block to zero, which is required by this call.

Line 20 fills in the **ioRefNum** field of the parameter block with the reference number for the .XPP driver that was received from the **OpenDriver** call.

Lines 21 and 22 assign the **csCode** and **xppSubCode** fields with the appropriate values for making the **GetMyZone** call.

Lines 23 and 24 fill in values that are passed along to the underlying ATP calls made by the .XPP driver on your behalf. These values tell it how long the ATP **SendRequest** should wait for retries and how many retries it should perform. See Chapter 7 for further information about ATP.

Line 25 assigns the address of the variable **returnedZoneName** to the **zipBuffPtr** field in the parameter block. **returnedZoneName** must be 33 bytes long—long enough to contain a 32-byte Pascal string—the maximum size of a zone name.

Line 28 makes the call to **PBControl** synchronously and line 30 assigns the returned zone name to the result of the function. If there is an error, the previously assigned value of '*' will be returned.

▶ Getting the Zone List

In order to build a list of the zones in a Phase 2 AppleTalk network, you need to make a call to **PBControl** with the parameter block's **csCode** field set to 246 and the **xppSubCode** field set to 6.

Listing 6-5 shows a function that gets the local zone using the PBControl call to the .XPP driver.

Listing 6-5. Getting the zone list

```
PROCEDURE BuildZoneList;
CONST
    kZonesSize = 578;
VAR
  theXPBPB                : xCallParam;
  zonePtr, currZonePtr    : Ptr;
  index, count            : INTEGER;
  xppDriverRefNum         : INTEGER;
  zoneName                : str255;
  stat                    : OSErr;
BEGIN {BuildZoneList}

  IF OpenDriver('.XPP', xppDriverRefNum) <> noErr
    THEN EXIT(BuildZoneList);

  zonePtr := NewPtr(kZonesSize);
  IF zonePtr = NIL
    THEN EXIT(BuildZoneList);

  WITH theXPBPB DO BEGIN
    zipInfoField[1] := 0;
    zipInfoField[2] := 0;
    zipLastFlag     := 0;

    ioRefNum    := xppDriverRefNum;
    csCode      := xCall;
    xppSubCode  := zipGetZoneList;
    xppTimeOut  := kATPTimeOutVal;
    xppRetry    := kATPRetryCount;
    zipBuffPtr  := zonePtr;
  END;

  index := 1;
  count := 0;

  REPEAT
```

```
37:     stat := PBControl(@theXPBPB, kSYNC);
38:     IF stat <> noErr THEN Leave;
39:
40:     count := count + theXPBPB.zipNumZones;
41:     currZonePtr := zonePtr;
42:     REPEAT
43:       zoneName := StrFromPtr(currZonePtr,currZonePtr^);
44:       ProcessZone(zoneName);
45:       currZonePtr := Ptr(LONGINT(currZonePtr) +
46:                                         currZonePtr^+1);
47:       index := index + 1;
48:     UNTIL index > count;
49:   UNTIL (theXPBPB.zipLastFlag <> 0);
50:   DisposPtr(zonePtr);
51: END; {BuildZoneList}
```

Lines 13 and 14 open the .XPP driver. This is a good way to get the reference number for the .XPP driver. If any error is encountered, the code exits the function without doing any further processing.

Line 16 allocates the 578 bytes of buffer space to store the zone information. Lines 17 and 18 exit if the memory couldn't be allocated.

Lines 21 and 22 set the first word in the **zipInfoField** field of the parameter block to zero, which is required by this call.

Line 23 clears the **zipLastFlag** field to zero.

Line 25 fills in the **ioRefNum** field of the parameter block with the reference number for the .XPP driver that was received from the **OpenDriver** call.

Lines 26 and 27 assign the **csCode** and **xppSubCode** fields with the appropriate values for making the **GetZoneList** call.

Lines 28 and 29 fill in values that are passed along to the underlying ATP calls made by the .XPP driver on your behalf. These values tell it how long the ATP **SendRequest** should wait for retries and how many retries it should perform. See Chapter 7 for further information about ATP.

Line 30 assigns the zone buffer, allocated in line 16, to the **zipBuffPtr** field of the parameter block.

Line 36 begins a loop that is repeated until the calls to **GetZoneList** tell you there are no more zones to get.

Line 37 makes the actual call to the **PBControl**, which is really the call **GetZoneList**. If the call returns an error condition, it exits the loop at line 38.

Line 40 increments the counter variable count by the number of zones that the call to **GetZoneList** returned.

Line 41 positions the **currZonePtr** to the beginning of the zone buffer in anticipation of the loop beginning at line 41. This loop repeats until each of the zones returned is processed.

Line 42 copies the zone name into a string so that line 43 can call a routine to do some processing based on that name.

Line 45 advances the pointer, **currZonePtr**, to point to the next zone name.

Line 47 increments the index so you can determine if you have finished processing the zone buffer in line 47 and exit the loop.

Line 49 tests for the **zipLastFlag**. When this is TRUE, it indicates that there are no more zone buffers to retrieve and process.

Line 50 cleans up the zone buffer that was allocated earlier.

▶ Getting the List of Local Zones

Because Phase 2 AppleTalk allows more than one zone to be assigned to a given local network, there needs to be a way for you to get this list. The **GetLocalZones** call is provided for this purpose.

The **GetLocalZones** call functions in a way very similar to the **GetZoneList** call. In fact, the same algorithm can be used to access it. The only difference is that you would make the **PBControl** call with the **xppSubCode** field of the parameter block set to 5 rather than 6.

Rather than list a code fragment to show how this is done, simply refer to Listing 6-5 and change the assignment of the **xppSubCode** field on line 27 to be **zipGetLocalZones** 5, rather than **zipGetZoneList**.

▶ Detailed Descriptions of Important ZIP Routines

The following section describes each important ZIP routine discussed previously in this chapter. It is restricted to the Phase 2 calls, because the Phase 1 "calls" aren't really calls, but rather ATP transactions. It shows the routine's prototype, lists all parameters or parameter block fields, and lists error codes. Each parameter or parameter block field and error code is then described in detail.

▶ GetMyZone

GetMyZone is used to get the name of the zone of the node on which your application is currently executing. The **GetMyZone** routine follows.

```
FUNCTION  PBControl(theXPPPBPtr:XPBPBPtr;  async:  BOOLEAN):OSErr;

Using  the  following  fields  in  the  XP  Parameter  Block:

     →       csCode                  — always xCall
     →       xppSubCode              — always zipGetMyZone
     ⇄       ioCompletion            — address of completion routine
     ←       ioResult                — result of operation
     →       xppTimeOut              — retry interval for the ATP call
     →       xppRetry                — retry count for the ATP call
     →       zipBuffPtr              — pointer to the zip buffer
     →       zipInfoField            — buffer space used by zip

Errors  Returned:

     noErr  (0)                      No error.
     noBridgeErr  (-93)              No router is present.
     reqFailed  (-1096)              Too many concurrent requests.
     tooManyReq  (-1097)             Too many outstanding ATP calls.
     noDataArea  (-1104)             No data area for request to MPP.
```

csCode always contains **xCall**, which is the constant 246.

xppSubCode always contains **zipGetMyZone**, which is the constant 7.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

xppTimeout and **xppRetry** determine the retry behavior of the underlying ATP transaction. **xppRetry** tells the ATP transaction how many retries should be attempted and **xppTimeout** tells it how long to wait between retries in 8–tick units.

zipBuffPtr contains a pointer to a buffer that will contain the zone name. This must be at least 33 bytes in size as the zone name is of the type Str32.

zipInfoField is a 70 byte buffer used by ZIP. The first two bytes of **zipInfoField** must be set to zero before making the call.
noErr is returned when the trap completes normally.
noBridgeErr is returned when there is no router present in the network.
reqFailed is returned when the retry count is exceeded before a valid response is completely received in the underlying ATP transaction.
tooManyReq is returned when there have been too many concurrent ATP requests made.
noDataArea is returned when AppleTalk runs out of memory to hold transaction information.

▶ GetZoneList

GetZoneList is used to retrieve the list of all known zones from a local router. It must be repeatedly called until it indicates there is no more zone information to return. The **GetZoneList** routine follows.

```
FUNCTION  PBControl(theXPPPBPtr:XPBPBPtr;  async:  BOOLEAN):OSErr;

Using the following fields in the XP Parameter Block:


    →      csCode            - always xCall
    →      xppSubCode        - always zipGetZoneList
    →      ioCompletion      - address of completion routine
    ←      ioResult          - result of operation
    →      xppTimeOut        - retry interval for the ATP call
    →      xppRetry          - retry count for the ATP call
    →      zipBuffPtr        - pointer to the zip buffer
    →      zipInfoField      - buffer space used by zip
    ←      zipNumZones       - number of zones gotten
    ←      zipLastFlag       - set to nonzero when no more zones

Errors Returned:

    noErr  (0)                No error.
    noBridgeErr  (-93)        No router is present.
    reqFailed  (-1096)        Too many concurrent requests.
    tooManyReq  (-1097)       Too many outstanding ATP calls.
    noDataArea  (-1104)       No data area for request to MPP.
```

csCode always contains **xCall**, which is the constant 246.

xppSubCode always contains **ZipGetZoneList**, which is the constant 6.

ioCompletion contains that address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

xppTimeout and **xppRetry** determine the retry behavior of the underlying ATP transaction. **xppRetry** tells it how many retries should be attempted and **xppTimeout** tells it how long to wait between retries in 8–tick units.

zipBuffPtr contains a pointer to a buffer that will contain the zone name information. This must be at least 578 bytes in size—large enough to contain an entire ATP packet.

zipInfoField is a 70 byte buffer used by ZIP. The first 2 bytes of **zipInfoField** must be set to zero before making the call.

zipNumZones contains the number of zones received by this call.

zipLastFlag contains a flag that indicates if there is more zone information to be gotten. It is either zero if there is more information to get or nonzero when there is no more.

noErr is returned when the trap completes normally.

noBridgeErr is returned when there is no router present in the network.

reqFailed is returned when the retry count is exceeded before a valid response is completely received in the underlying ATP transaction.

tooManyReq is returned when there have been too many concurrent ATP requests made.

noDataArea is returned when AppleTalk runs out of memory to hold transaction information.

▶ GetLocalZones

GetLocalZones is used to retrieve the list of all the local zones for the local network. It must be repeatedly called until it indicates there is no more local zone information to return. The **GetLocalZones** routine follows.

```
FUNCTION  PBControl(theXPPPBPtr:XPBPBPtr;  async:
BOOLEAN):OSErr;

Using the following fields in the XP Parameter Block:

    →      csCode              - always xCall
    →      xppSubCode          - always zipGetLocalZones
    →      ioCompletion        - address of completion routine
    →      ioResult            - result of operation
    →      xppTimeOut          - retry interval for the ATP call
    →      xppRetry            - retry count for the ATP call
    →      zipBuffPtr          - pointer to the zip buffer
    →      zipInfoField        - buffer space used by zip
    ←      zipNumZones         - number of zones gotten
    ←      zipLastFlag         - set to nonzero when no more zones

Errors Returned:

    noErr  (0)                 No error.
    noBridgeErr  (-93)         No router is present.
    reqFailed  (-1096)         Too many concurrent requests.
    tooManyReq  (-1097)        Too many outstanding ATP calls.
    noDataArea  (-1104)        No data area for request to MPP.
```

csCode always contains **xCall**, which is the constant 246.

xppSubCode always contains **zipGetLocalZones**, which is the constant 5.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

xppTimeout and **xppRetry** determine the retry behavior of the underlying ATP transaction. **xppRetry** tells it how many retries should be attempted and **xppTimeout** tells it how long to wait between retries in 8-tick units.

zipBuffPtr contains a pointer to a buffer that will contain the zone name information. This must be at least 578 bytes in size—large enough to contain an entire ATP packet.

zipInfoField is a 70 byte buffer used by ZIP. The first two bytes of **zipInfoField** must be set to zero before making the call.

zipNumZones contains the number of zones received by this call.

zipLastFlag contains a flag that indicates if there is more zone information to be gotten. It is either zero if there is more information to get or nonzero when there is no more.

noErr is returned when the trap completes normally.

noBridgeErr is returned when there is no router present in the network.

reqFailed is returned when the retry count is exceeded before a valid response is completely received in the underlying ATP transaction.

tooManyReq is returned when there have been too many concurrent ATP requests made.

noDataArea is returned when AppleTalk runs out of memory to hold transaction information.

▶ Summary

This chapter covered the Zone Information Protocol. It showed how ZIP can be used to get zone information. Specifically, it showed how to get your local zone name, the list of all zones, and the list of all local zones.

This chapter also showed how ZIP can be used in both Phase 1 and Phase 2 AppleTalk environments using ATP transactions or calls to the .XPP driver.

Chapter 7 covers the AppleTalk Transaction Protocol. Details of making and receiving requests are discussed along with sending responses and aborting calls. Specific ATP routines follow the discussions.

7 ▶ AppleTalk Transaction Protocol

AppleTalk Transaction Protocol (ATP) is a basic component of AppleTalk. It provides you with a simple way to reliably transfer relatively small amounts of data across AppleTalk networks.

ATP makes use of the basic idea of a transaction to move data across the network. This is an asymmetric operation where on one side, the requester makes a request, and on the other side, the responder responds. These transactions are limited in the amount of data that can be transferred, but it is easy to work around this limitation by performing multiple transactions when larger amounts of data are to be transferred across the network.

Note ▶ Although ATP can be made to transfer large amounts of data using multiple transactions, you should also explore the use of AppleTalk Data Stream Protocol (ADSP) when large amounts of data must be transferred because it is optimized for large data transfers.

There are many uses for ATP. AppleTalk itself uses it to implement the Zone Information Protocol (ZIP) as well as Printer Access Protocol (PAP) and AppleTalk Session Protocol (ASP). Any application where a program needs to make a request of a remote program should consider using ATP.

▶ The Mechanics of ATP Transactions

ATP transactions provide reliable service. That is, the transaction will complete properly or an error will be returned to you. All of the details dealing with retransmission of dropped or lost packets are handled by this protocol. ATP does its best to complete the transaction for you.

Even though ATP takes care of the details of implementing a transaction, some understanding of how a transaction works is helpful to make the best use of ATP. ATP provides a number of controls over the details of a transaction and the proper utilization of these controls can make your program more efficient.

ATP's basic model for a transaction is that first the requester sends out a request packet. Then the responder receives the request packet and does some processing to prepare a response. The response is then sent back to the receiver in up to eight response packets. These packets are assembled into a response message and delivered to the original requester. This process is illustrated in Figure 7-1.

There are some obvious limitations with this scheme. Clearly, ATP transactions are asymmetric—more data can be transferred in the response than in the request.

Also, because ATP is layered on top of DDP and ATP packets are layered inside of DDP packets, ATP packets can only contain up to 578 data bytes. DDP packets have 586 data bytes and ATP uses 8 of these bytes as a header. This means that the request can be up to 578 bytes long and the response can be up to 4624 bytes long. To transfer more than these amounts, multiple transactions are required.

When a transaction does not complete, ATP will retry it a number of times. This is controlled by two fields in the parameter block used in the

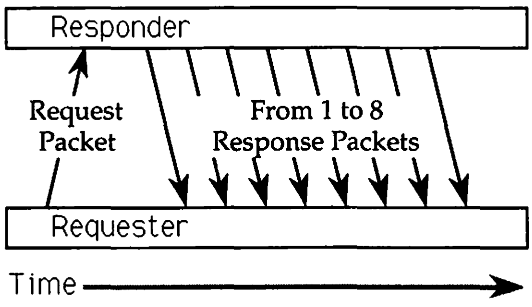


Figure 7-1. Normal ATP transaction

PSendRequest call: **timeOutVal** and **retryCount**. The **timeOutVal** determines how long ATP waits before resending the original request packet. The number of times this is done is controlled by the **retryCount** field. So the total retry time is **timeOutVal** (in seconds) times the retry count.

One important optimization that ATP makes when performing retries has to do with the response bitmap that it maintains in the ATP packet header. This bitmap (8 bits, one for each possible response packet) is used in the request packet to tell the responder which response packets have not been received by the requester. (Note that the response packets are numbered from zero to seven.) In the case of retries, this bitmap tells the responder which packets have already been safely received by the requester. The responder only has to resend the packets that have not been received and not bother to send the others. Figure 7-2 shows how this works. It shows a request being made that expects four reply packets. These packets are sent by the responder. Packet 2 is lost during transmission. After the timeout period is up, the requester resends its request packet with a bitmap telling the responder that it still needs packet 2. This is then sent by the responder to complete the transaction.

Another common scenario is to have the responder disappear during a transaction (it gets turned off, crashes, or is disconnected from the network). In this case, AppleTalk will wait for the retry period, then resend the request. It repeats this **retryCount** the number of times specified in the **retryCount** field of the parameter block. After exhausting the retries, AppleTalk gives up and returns an error of **reqFailed** (–1096) telling you that the retry count has been exceeded.

Choosing the proper combination of **timeOutVal** and **retryCount** is very specific to your application. If your program will be used by a variety of users on a variety of networks, make sure you allow generous values.

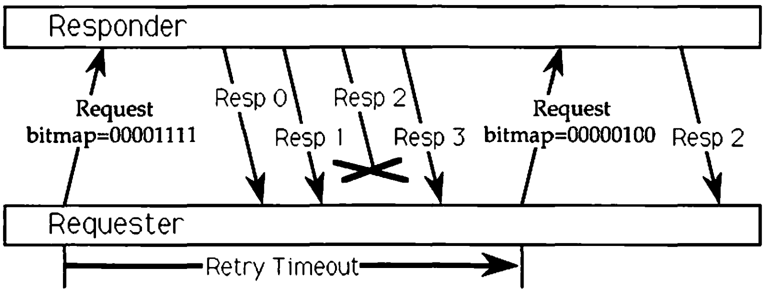


Figure 7-2. ATP transaction with lost response

Large networks often have large delays caused by multiple routers, network congestion, or slow links (such as modems). Values for these parameters that work well in a small test network are often too restrictive for larger networks.

Sometimes it can be good to allow power users to directly set these values, though remember that most users will have no idea what these parameters mean. A better approach can be to send some test transactions to estimate network performance and adapt your values to this. You may also use the AppleTalk Echo Protocol (AEP) which is discussed in Chapter 2.

There are times when you want ATP to keep retrying indefinitely. For this type of situation, AppleTalk treats a **retryCount** of 255 as meaning retry forever. With this setting, AppleTalk will continue retrying over and over again until either the transaction completes or it is explicitly canceled.

The use of the bitmap for lost packet recovery works well, but it depends on the requester asking for a certain number of packets and the responder sending that many packets back. There are times when the responder does not wish to send a response with as many packets as the requester asked for. This case is handled by setting the **end-of-message** flag in the response packet.

Another control you have over ATP is choosing either at-least-once transactions or exactly-once transactions. *At-least-once* transactions are the most efficient type of ATP transaction. They guarantee that a given request is received by the responder at least once. This means that it could be received, and acted upon by the responder, more than once. This mode of operation is appropriate for transactions that don't have any side effects on the responder side. For example, getting a status from a remote machine often works this way. You specify an at-least-once transaction by clearing the **XO** flag in the **atpFlags** field of the ATP parameter block. Figure 7-3 illustrates this process. A request is sent and received. The response is lost for any of a number of reasons, so after the timeout period the request is resent. This request is again received by the responder and acted upon. The second response is sent and received by the requester.

Exactly-once transactions have more overhead than at-least-once transactions, but they guarantee that the responder only receives a given request once. AppleTalk does this by saving the response packets until the transaction is complete. This means that AppleTalk can retransmit packets itself without the intervention of your code.

Of course there must be a way to free up the saved response packets. This is done with an additional packet called a *transaction release packet*. The transaction release packet is sent by the requester when it has received all the response packets it expects from the responder. If for any reason the transaction release packet is lost, AppleTalk uses another mechanism to

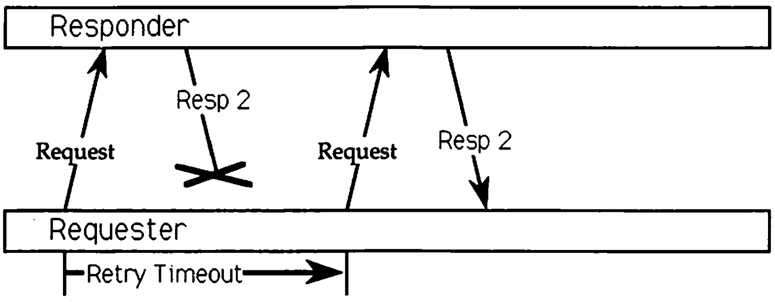


Figure 7-3. ATP transaction with XO option

free up the saved response packets. This mechanism is the *transaction release timer*. Each packet saved for possible retransmission is timestamped when it is saved. If more than the release time passes, the packet is discarded and the space consumed by the packet is freed.

The release time is fixed at 30 seconds in AppleTalk Phase 1. This has been expanded in AppleTalk Phase 2 to allow the programmer to select from five possible release times: 30 seconds, 1 minute, 2 minutes, 4 minutes, or 8 minutes. You specify these release times by setting the last 3 bits of the **atpFlags** field of the ATP parameter block using the following values:

000 (0) for 30 seconds
001 (1) for 1 minute
010 (2) for 2 minutes
011 (3) for 4 minutes
100 (4) for 8 minutes

Note ▶ **These values are ignored by AppleTalk Phase 1, which always uses a release time of 30 seconds.**

▶ The Buffer Data Structure

The Buffer Data Structure (BDS) exists to handle ATP responses that are made up of multiple response packets, which can be returned in any order. AppleTalk needs some way to manage these returning packets and the BDS does this.

The basic structure of the BDS is an array of eight records, one for each returning packet. Each of these records, known as a **BDSElement**, is structured as shown in Listing 7-1. The **BDSElement** contains a pointer to a buffer where the data from the response packet is actually stored along with a size for that buffer. It also contains the actual size of the data received in the packet and the user bytes from the packet.

Listing 7-1. Structure of a BDS element

```
TYPE BDSElement = RECORD
    buffSize      : INTEGER;   { Size of buffer in bytes }
    buffPtr       : PTR;       { Pointer to buffer }
    dataSize      : INTEGER;   { Actual size of data received }
    userBytes     : LONGINT;   { User bytes received in packet }
END; {BDSElement}
```

It is possible to set up and manage your own BDS, but the AppleTalk Manager provides a routine to do this for you that is often more convenient to use. This routine is called **BuildBDS**. To use **BuildBDS** you pass it a pointer to your response message buffer along with the size of this buffer, as well as a pointer to your BDS. Remember that your entire message cannot be more than 4624 bytes long.

Returned as the value of **BuildBDS** is the number of actual BDS elements required to contain your message. For example, if your response message buffer is the maximum 4624 bytes long, **BuildBDS** will return 8 since this length of message requires the full eight possible elements to manage it. A shorter response message, say 100 bytes long, would only require one element to manage, so **BuildBDS** would return 1.

▶ Using AppleTalk Transaction Protocol

There are two basic sides to using ATP: acting as a requester and acting as a responder. There are a number of variations on these basic operations and descriptions of these variations follow.

▶ Making a Request

To send an ATP request, you use the **PSendRequest** trap. This trap sends your data to another ATP socket and waits for the response to come back.

Listing 7-2 shows the code for a simple routine named **SendStrRequest** that shows how ATP requests are sent. ATP sends a single string to a remote ATP socket, then receives a single string back as its response. It has the following three parameters:

1. A destination address where the message is to be sent
2. A string to be sent to the remote location
3. A response string that should come back from the remote program

Listing 7-2. SendStrRequest routine using PSendRequest

```
PROCEDURE SendStrRequest(    destination : AddrBlock;
                             sendString  : str255;
                         VAR respString  : str255);
VAR
    myBDS    : BDSType;
    stat     : OSErr;
    myATPPB  : ATPParamBlock;
    BDSCount : INTEGER;
BEGIN {SendStrRequest}

    BDSCount := BuildBDS(@respString,@myBDS,
                            SIZEOF(respString));

    WITH myATPPB DO BEGIN
        bdsPointer      := @myBDS;
        numOfBuffs      := BDSCount;
        addrBlock       := destination;
        timeOutVal      := kATPTimeOutVal;
        retryCount      := kATPRetryCount;
        atpFlags        := atpXOvalue;
        reqLength       := SIZEOF(sendString);
        reqPointer      := @sendString;
        ioCompletion    := NIL;
    END;

    IF PSendRequest(@myATPPB,kSYNC) <> noErr
        THEN BEGIN
            respString := 'ERROR';
        END;

END; {SendStrRequest}
```

The first thing **SendStrRequest** does is build its BDS at line 11. The message coming back is a simple response string, so you pass a pointer to the response string variable and its size. The finished BDS is built using **myBDS**, and also passed via a pointer. The number of BDS elements used by **BuildBDS** is returned by **BuildBDS** into the variable **BDSCount**.

Next, **SendStrRequest** fills in the needed fields of the ATP parameter block. Lines 15 and 16 deal with the response BDS. Line 15 assigns the address of **myBDS** to the **bdsPointer** field and line 16 assigns the **BDSCount** returned by **BuildBDS** to the **numOfBuffs** field.

Line 17 puts the address of the remote ATP socket you wish to talk to into the **addrBlock** field. This address is typically found using an **NBPLookup**, though this is not shown in this example.

Lines 18 and 19 fill in the timeout and retry values used by this transaction. Here global constants are used.

Line 20 specifies that the exactly-once mode of ATP should be used by setting the **XO** flag in the **atpFlags** field.

Lines 21 and 22 specify the outgoing message. In this case it is simply the string passed into **SendStrRequest** so you assign the length of the string data to the **reqLength** field and the address of the string to the **reqPointer** field. Note that you use the length of the entire string as the **reqLength**. You could also have passed the length of the string itself (plus 1 for the length byte). This would reduce the message size and lead to a faster transmission time.

After sending the message at line 26 asynchronously, either the response is returned, or an error condition is flagged in the result returned from **PSendRequest**.

If there is no error, the incoming message will have already been stored in the **respString** parameter because you built the BDS using the **respString** parameter as its storage.

The example in Listing 7-2 doesn't utilize the user bytes. User bytes can be set to anything you wish for such things as command information or additional data you don't want placed in your main message. You send user bytes by simply assigning a long word to the **userData** field of the parameter block before sending the message.

Note that the example in Listing 7-2 does use the synchronous mode of operation. This keeps the example simple, but it isn't very often that a synchronous **PSendRequest** is used in real life. Most of the time you need to operate asynchronously, using either one of the polling techniques, or a completion routine. See Chapter 3 for further details.

▶ Using PNSendRequest Rather than PSendRequest

There is another call provided to make ATP requests; this is **PNSendRequest**. It is a variation of the **PSendRequest** call that pays attention to one additional field in the ATP parameter block. **PNSendRequest** looks at the **atpSocket** field and uses the specified socket for the ATP transaction. The specified socket must have been opened for use by ATP using the **POpenATPSocket** call.

The **PSendRequest** call dynamically opens an ATP socket for each request it sends, then closes this socket when the request completes. Using **PNSendRequest**, you can open a single ATP socket and use it for several transactions. This eliminates the slight overhead of opening and closing a socket for each transaction.

▶ Receiving a Request

To receive an ATP request you use the **PGetRequest** trap. This trap waits for an ATP request to come in on the specified socket. 99 percent of the time, you will want to use this call asynchronously—sitting around waiting for ATP requests is a very unpopular pastime for most users.

Listing 7-3 shows a fragment of code that makes an asynchronous **PGetRequest**. This listing illustrates receiving a message defined by the type **PacketRec** with a buffer allocated of that type called **incomingPacket**.

Listing 7-3. Making an asynchronous PGetRequest

```
1: WITH incomingATPPB DO BEGIN
2:     reqLength    := SIZEOF(PacketRec);
3:     reqPointer   := @incomingPacket;
4:     atpSocket    := mySocket;
5:     ioCompletion := NIL;
6: END;
7:
8: stat := PGetRequest(@incomingATPPB,kASYNC);
```

Line 2 assigns the size of the expected incoming packet to the **reqLength** field of the ATP parameter block. No more data will be accepted than the amount specified here.

Line 3 assigns the address of the incoming packet buffer to the **reqPointer** field of the ATP parameter block. This tells the AppleTalk Manager where to store the incoming data.

Line 4 assigns the socket address stored in **mySocket** to the **atpSocket** field. This socket must have been previously opened using **POpenATPSocket**.

Finally, line 5 assigns NIL to the **ioCompletion** routine field of the ATP parameter block. This indicates to the AppleTalk Manager that no completion routine should be called upon completion of the request. This, together with specifying in line 8 that asynchronous mode should be used, allows us to poll for an actual incoming request.

Once you have issued the asynchronous **PGetRequest**, you need to poll for the completion of the routine since no **ioCompletion** routine was specified. See the Example RDEV code in Chapter 11 for an illustration of using a completion routine with **PGetRequest**. Listing 7-4 shows a code fragment that checks for the completion of the **PGetRequest** call made in Listing 7-3.

Listing 7-4. Completion polling of PGetRequest

```
1: IF incomingATPPB.ioResult <> 1 THEN
2:   IF incomingATPPB.ioResult = noErr THEN
3:     ProcessIncoming(incomingATPPB)
4:   ELSE
5:     HandleError(incomingATPPB.ioResult);
```

Line 1 checks for the completion of the **PGetRequest**. The call is completed for one of two reasons: either a request came in or some error occurred. Line 2 checks this and if all is well, line 3 calls a routine that will process the incoming packet. Otherwise, line 5 calls an error handling routine.

▶ Sending a Response

Once an ATP request has been received, it needs a response. The **PSendResponse** call serves this function. Many of the fields in the ATP parameter block required by **PSendResponse** are the same as those that are filled in by **PGetRequest**. These fields should be copied from the parameter block used by **PGetRequest** into the parameter block used by **PSendResponse**. These fields include **addrBlock**, **reqTID**, and **atpSocket**; however, there may be times when you wish to use a socket other than the one the request came in on to send back the response. In that case, you could fill in **atpSocket** with any socket that has been opened with **POpenATPSocket**.

Listing 7-5 shows a code fragment for issuing a response to a request. It begins by constructing a BDS for the outgoing message using **BuildBDS** on line 1.

Listing 7-5. Responding to a Request

```
 1: BDSCount := BuildBDS(@outPacket,@myBDS,
 2:                      SIZEOF(respString));
 3: outgoingATPPB := incomingATPPB;
 4: WITH outgoingATPPB DO BEGIN
 5:   atpFlags     := atpEOMvalue;
 6:   bdsPointer   := @outgoingBDS;
 7:   bdsSize      := BDSCount;
 8:   numOfBuffs   := BDSCount;
 9:   ioCompletion := NIL;
10: END;
11: stat := PSendResponse(@outgoingATPPB,kASYNC);
```

Line 3 copies the entire incoming parameter block into the outgoing parameter block. This fills in the **addrBlock, reqTID,** and **atpSocket** fields for us.

Line 5 sets the **atpFlags** field to the constant value **atpEOMvalue,** which denotes that this includes the end of message.

Lines 6 assigns the **bdsPointer** field to point to the beginning of the outgoing BDS prepared in lines 1 and 2.

Lines 7 and 8 assign **BDSCount,** the number of BDS elements returned by the **BuildBDS** call in lines 1 and 2, to both the **bdsSize** and the **numOfBuffs** fields. These almost always are set to the same value.

Finally, line 11 makes an asynchronous call to **PSendResponse** to have the response actually sent as you continue processing.

▶ Aborting ATP Calls

There are times when you need to cancel an ATP call that you have previously issued. There are two basic ways to accomplish this: closing the socket, or using the provided "kill" call.

Closing a socket will abort any outstanding operation using that socket. This is often the easiest way to abort everything when you are finished with a socket.

A more precise way to abort specific calls is to use **PKillSendRequest** or **PKillGetRequest.** As their names imply, **PKillSendRequest** aborts a **PSendRequest** or a **PNSendRequest** call and **PKillGetRequest** aborts a **PGetRequest** call.

Both **PKillSendRequest** and **PKillGetRequest** take as their parameters the address of the parameter block used by the specific request you are sending or getting.

▶ Detailed Descriptions of Important ATP Routines

The following section describes each important ATP routine. It shows the routine's prototype, lists all parameters or parameter block fields, and lists error codes. Each parameter or parameter block field and error code is then described in detail.

▶ PSendRequest

PSendRequest is used to send an ATP request to another program. It completes operation after either receiving a response message or encountering an error condition such as a timeout. The **PSendRequest** routine follows.

```
FUNCTION PSendRequest(thePBPtr:ATPPBPtr; async:
BOOLEAN):OSErr;

Using the following fields in the ATP Parameter Block:

 →  ioCompletion        – address of completion routine
 ←  ioResult            – result of operation
 →  timeOutVal          – retry interval in eight tick units
 ↔  retryCount          – retry count
 →  userData            – user bytes
 ←  reqTID              – transaction ID used in request
 ←  atpSocket           – current bitmap
 ↔  atpFlags            – control information
 →  addrBlock           – destination socket address
 →  reqLength           – request size
 →  reqPointer          – pointer to request data
 →  bdsPointer          – pointer to response BDS
 →  numOfBuffs          – number of response packets expected
 ←  numOfResps          – number of response packets received

Errors Returned:

    noErr (0)                No error.
    reqFailed (-1024)        Retry count exceeded.
    tooManyReqs   (-1096)    Too many concurrent requests.
    noDataArea (-1104)       Too many outstanding ATP calls.
    reqAborted (-1105)       Request canceled by user.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

timeOutVal and **retryCount** determine the retry behavior of the trap. **retryCount** tells it how many retries should be attempted and **timeOutVal** tells it how long to wait between retries in 8–tick units. **retryCount** is decremented with each retry.

userData contains 4 bytes of data that are sent with the message in its header. They can be used for any purposes the user wishes.

reqTID is a number that identifies the current transaction. It can be used later in **PNKillSendRequest** to abort the transaction.

atpSocket returns the bitmap showing which response packets were actually received before completion. This can be used to recover partial data and assist in error recovery.

atpFlags contains the ATP flag values. You set the XO bit (bit 5) to use an exactly-once transaction, or clear it to use an at-least-once transaction. When using an exactly-once transaction, you set the release timer by setting the low 3 bits (bits 0-2) with the values shown in "The Mechanics of ATP Transactions" earlier in this chapter; when not using an exactly-once transaction, these bits should be set to zero.

addrBlock contains the address of the responder's socket.

reqLength and **reqPointer** describe the location and size of the data that should be sent to the responder. It must not be larger than 578 bytes in size.

bdsPointer points to the response BDS that will contain the response data.

numOfBuffs contains the number of response packets expected from the responder.

numOfResps contains the number of response packets actually received from the responder.

noErr is returned when the trap completes normally.

reqFailed is returned when the retry count is exceeded before a valid response is completely received.

tooManyReqs is returned when the number of concurrent ATP requests exceeds the maximum. The maximum number of ATP requests is determined by the version of AppleTalk running along with which CPU you are running on.

noDataArea is returned when AppleTalk runs out of memory to hold transaction information.
reqAborted is returned when the trap is aborted by the user using the **PKillSendRequest** trap.

▶ PNSendRequest

PNSendRequest is identical to **PSendRequest** with one difference: Rather than dynamically allocating a socket to use for the transaction, it uses the socket specified in the **atpSocket** field of the parameter block. The **PNSendRequest** routine follows.

```
FUNCTION PNSendRequest(thePBPtr:ATPPBPtr; async:
BOOLEAN):OSErr;

Using the following fields in the ATP Parameter Block:

 →  ioCompletion       - address of completion routine
 ←  ioResult           - result of operation
 →  timeOutVal         - retry interval in eight tick units
 ↔  retryCount         - retry count
 →  userData           - user bytes
 ←  reqTID             - transaction ID used in request
 ↔  atpSocket          - socket to use / current bitmap
 ↔  atpFlags           - control information
 →  addrBlock          - destination socket address
 →  reqLength          - request size
 →  reqPointer         - pointer to request data
 →  bdsPointer         - pointer to response BDS
 →  numOfBuffs         - number of response packets expected
 ←  numOfResps         - number of response packets received
 ←  intBuff            - internal use

Errors Returned:

    noErr (0)                 No error.
    reqFailed (-1024)         Retry count exceeded.
    tooManyReqs  (-1096)      Too many concurrent requests.
    noDataArea (-1104)        Too many outstanding ATP calls.
    reqAborted (-1105)        Request canceled by user.
    badATPSocket (-1099)      Socket does not exist.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

timeOutVal and **retryCount** determine the retry behavior of the trap. **retryCount** tells it how many retries should be attempted and **timeOutVal** tells it how long to wait between retries in 8–tick units. **retryCount** is decremented with each retry.

userData contains 4 bytes of data that are sent with the message in its header. They can be used for any purposes the user wishes.

reqTID is a number that identifies the current transaction. It can be used later in **PNKillSendRequest** to abort the transaction.

atpSocket is used to pass in the socket number that ATP should use for the transaction. This same parameter also returns the bitmap showing which response packets were actually received before completion. This can be used to recover partial data and assist in error recovery.

atpFlags contains the ATP flag values. You can either set the XO bit (bit 5) to use an exactly-once transaction, or clear it to use an at-least-once transaction. When using an exactly-once transaction you set the release timer by setting the low 3 bits (bits 0-2) with the values shown in "The Mechanics of ATP Transactions" earlier in this chapter; when not using an exactly-once transaction, these bits should be set to zero.

addrBlock contains the address of the socket of the responder.

reqLength and **reqPointer** describe the location and size of the data that should be sent to the responder. It must not be larger than 578 bytes in size.

bdsPointer points to the response BDS that contains the response data.

numOfBuffs contains the number of response packets expected from the responder.

numOfResps contains the number of response packets actually received from the responder.

intBuff is an internal buffer used by ATP.

noErr is returned when the trap completes normally.

reqFailed is returned when the retry count is exceeded before a valid response is completely received.

tooManyReqs is returned when the number of concurrent ATP requests exceeds the maximum. The maximum number of ATP requests is determined by the version of AppleTalk running along with which CPU you are running on.

noDataArea is returned when AppleTalk runs out of memory to hold transaction information.

reqAborted is returned when the trap is aborted by the user using the **PKillSendRequest** trap.

badATPSocket is returned when the specified socket is invalid.

▶ PGetRequest

PGetRequest is used to ask ATP for any incoming request messages. It is almost always called asynchronously. The **PGetRequest** routine follows.

```
FUNCTION PGetRequest(thePBPtr:ATPPBPtr; async: BOOLEAN):OSErr;

Using the following fields in the ATP Parameter Block:

 →  ioCompletion        - address of completion routine
 ←  ioResult            - result of operation
 →  userData            - user bytes
 ←  reqTID              - transaction ID used in request
 ↔  atpSocket           - socket to use
 ↔  atpFlags            - control information
 →  addrBlock           - destination socket address
 →  reqLength           - request size
 →  reqPointer          - pointer to request data
 →  bitMap              - transaction bitmap

Errors Returned:

    noErr (0)                    No error.
    reqAborted (-1105)           Request canceled by user.
    badATPSocket (-1099)         Socket does not exist.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

userData contains 4 bytes of data from the message header. They can be used for any purposes the user wishes.

reqTID is a number that identifies this transaction. It is used later in the **PSendResponse** trap.

atpSocket is the socket where ATP should look for an incoming request.

atpFlags contains the ATP flag values.

addrBlock contains the address of the socket of the requester.

reqLength and **reqPointer** describe the location and size of the request data. This buffer should be 578 bytes long (the maximum size of a request packet), unless you are sure that the request will be shorter than this.

bitMap contains the transaction bitmap. This can be used to ascertain how many packets the requester expects as a response.

noErr is returned when the trap completes normally.

reqAborted is returned when the trap is aborted by the user using the **PKillSendRequest** trap.

badATPSocket is returned when the specified socket is invalid.

▶ PSendResponse

PSendResponse is used to send the response message back to the requester. It is called after a request has been received and a response message has been created. The **PSendResponse** routine follows.

```
FUNCTION PSendResponse(thePBPtr:ATPPBPtr; async:
BOOLEAN):OSErr;

Using the following fields in the ATP Parameter Block:

→    ioCompletion       - address of completion routine
←    ioResult           - result of operation
→    userData           - user bytes
←    reqTID             - transaction ID used in request
↔    atpSocket          - socket to use
↔    atpFlags           - control information
→    addrBlock          - destination socket address
→    bdsPointer         - pointer to response BDS
→    bdsSize            - size of the BDS
→    numOfBuffs         - number of response packets

Errors Returned:

     noErr (0)                    No error.
     badATPSocket (-1099)         Socket does not exist.
     noRelErr (-1101)             No release received.
     noDataArea (-1104)           Too many outstanding ATP calls.
     badBuffNum (-1100)           Bad response buffer number.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

userData contains 4 bytes of data that are sent with the message in its header. They can be used for any purposes the user wishes.

reqTID is a number that identifies this transaction. It should be copied from the incoming request.

atpSocket is the socket to use to send the response.

atpFlags contains the ATP flag values. You should set the end-of-message bit (bit 4) when you are sending fewer response packets than the requester expects.

addrBlock contains the address where the response should be sent.

bdsPointer points to the response BDS with enough space for the response packets.

bdsSize is the size of the response BDS in elements. This is normally the returned value from the **BuildBDS** call.

numOfBuffs contains the number of response packets to be sent with this call. This is usually the same as **bdsSize**.

noErr is returned when the trap completes normally.

badATPSocket is returned when the specified socket is invalid.

noRelErr is returned when no release message is received.

noDataArea is returned when AppleTalk runs out of memory to hold transaction information.

badBuffNum is returned when a bad buffer response number is specified.

▶ POpenATPSocket

POpenATPSocket is used to open an ATP socket for use later to either make a request using **PNSendRequest**, or to receive requests using **PGetRequest**. The **POpenATPSocket** routine follows.

```
FUNCTION POpenATPSocket(thePBPtr:ATPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ATP Parameter Block:

 →  ioCompletion        - address of completion routine
 ←  ioResult            - result of operation
 ↔  atpSocket           - socket number to open
 ←  addrBlock           - addresses to accept on this socket

Errors Returned:

     noErr (0)                  No error.
     tooManySockets  (-1098)    Too many sockets are already
                                open.
     noDataArea  (-1104)        Too many outstanding ATP calls.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.
atpSocket contains the socket to open. If this contains 0, ATP will assign any free socket and fill in this field with that socket number.
addrBlock contains an address that specifies which other sockets this socket will accept requests from. Fill this with zeros if you want to accept requests from any socket.
noErr is returned when the trap completes normally.
tooManySockets is returned when the ATP runs out of available sockets.
noDataArea is returned when ATP runs out of memory.

▶ PCloseATPSocket

PCloseATPSocket closes the specified socket. Any memory consumed by that socket is given up and any outstanding asynchronous operation pending on that socket is aborted and its **ioResult** field is set to **sktClosed**. The **PCloseATPSocket** routine follows.

```
FUNCTION  PCloseATPSocket(thePBPtr:ATPPBPtr;  async:  BOOLEAN):OSErr;

Using the following fields in the MPP Parameter Block:

 →  ioCompletion         - address of completion routine
 ←  ioResult             - result of operation
 ↔  atpSocket            - socket number to close

Errors Returned:

    noErr  (0)                 No error.
    noDataArea  (-1104)        Too many outstanding ATP calls.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.
ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.
atpSocket contains the socket to close.
noErr is returned when the trap completes normally.
noDataArea is returned when ATP runs out of memory.

▶ PKillSendRequest

PKillSendRequest is used to abort an outstanding request made by either **PGetRequest** or **PNGetRequest**. The **PKillSendRequest** routine follows.

```
FUNCTION  PKillSendReq(thePBPtr:ATPPBPtr;  async:  BOOLEAN):OSErr;

Using  the  following  fields  in  the  ATP  Parameter  Block:

 →  ioCompletion        –  address  of  completion  routine
 ←  ioResult            –  result  of  operation
 →  aKillQEl            –  control  block  of  call  to  abort

Errors  Returned:

    noErr  (0)                   No  error.
    cbNotFound  (-1102)          Control  block  not  found.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

aKillQEl contains a pointer to the *control block* (parameter block) of the SendReq or NSendReq trap that is to be aborted.

noErr is returned when the trap completes normally.

cbNotFound is returned when the specified pointer does not point to a valid **SendReq** or **NSendReq** parameter block.

▶ PKillGetRequest

PKillGetRequest is used to abort an outstanding PGetRequest call. The **PKillGetRequest** routine follows.

```
FUNCTION  PKillGetReq(thePBPtr:ATPPBPtr;  async:  BOOLEAN):OSErr;

Using  the  following  fields  in  the  ATP  Parameter  Block:

 →    ioCompletion           – address  of  completion  routine
 ←    ioResult               – result  of  operation
 →    aKillQEl               – control  block  of  call  to  abort

Errors  Returned:

    noErr  (0)                 No  error.
    cbNotFound  (-1102)        Control  block  not  found.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

aKillQEl contains a pointer to the control block (parameter block) of the GetReq trap that is to be aborted.

noErr is returned when the trap completes normally.

cbNotFound is returned when the specified pointer does not point to a valid **GetReq** parameter block.

▶ Summary

This chapter described the operation of ATP. It detailed the low-level operation of ATP and how this can affect your use of it. Examples of how to use the ATP routines in your own programs were then provided. Furthermore, each ATP routine that you would need to use was described. Each parameter or parameter block field used in every routine was discussed in detail.

Chapter 8 is about AppleTalk Data Stream Protocol. More specific routines will be given after discussions of connection, structure, the Data Stream, and attention messages.

8 ▶ AppleTalk Data Stream Protocol

The AppleTalk Data Stream Protocol (ADSP) provides a full-duplex data stream between any two sockets on an AppleTalk network. That is, it lets two programs send and receive continuous streams of data back and forth across the network. It also provides for out-of-band signaling via attention messages. These messages are sent between the two programs without disrupting the primary data stream.

ADSP provides a service that is in many ways analogous to serial communications. Serial communications involves sending continuous streams of data bidirectionally, as does ADSP. However, unlike serial communications, which sends its data as a continuous stream of bits down a wire, ADSP actually sends its data across the network in packets using DDP. In most circumstances, you don't have to concern yourself with the mechanics of how the data is accumulated into packets for transmission over DDP, but ADSP does provide some control over this for you when it is required.

▶ ADSP Connections

One of the basic concepts in ADSP is the connection. An *open connection* connects two sockets together and allows data and attention messages to flow freely across it. Before an open connection can be established, both ends must prepare for it by creating a connection end. The connection ends can then be combined into a fully operational open connection.

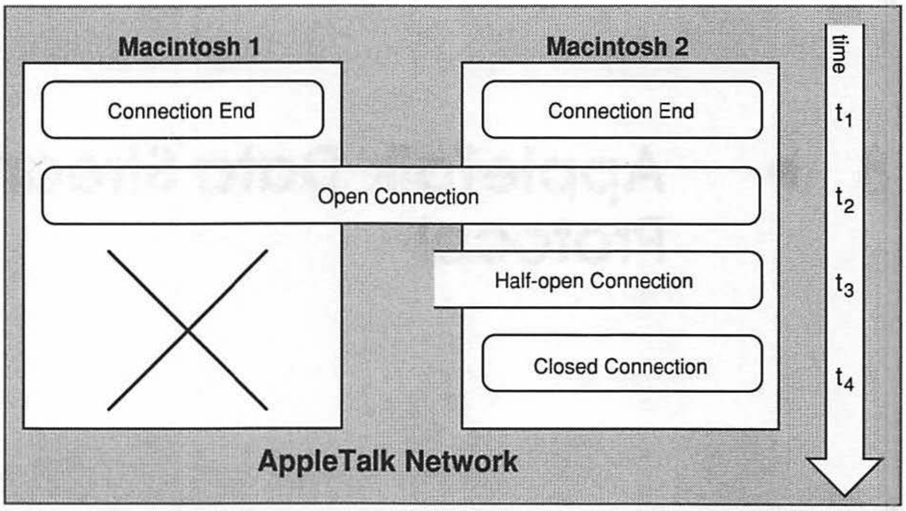


Figure 8-1. A typical life cycle of a connection

Once a connection is opened, it can be closed by either end. If one connection end loses contact with the other, the connection becomes half-open. ADSP will close a half-open connection after two minutes if the connection cannot be reestablished.

Figure 8-1 shows the life cycle of a typical connection. It begins at time t_1 as two connection ends on two separate Macintosh computers on an AppleTalk network. Then, at time t_2, the connection is made allowing the two ends to exchange data and attention messages (a section is devoted to attention messages later in this chapter). At time t_3, the second Macintosh loses contact with the first (maybe the first one has been powered off or has turned off AppleTalk) leaving a half-open connection. After two minutes, at time t_4, the half-open connection is closed.

► Making an ADSP Connection

There are two basic models for establishing connections in ADSP. The first and simplest model is shown in Figure 8-2. It involves opening one connection end using the passive mode and opening the other connection end using the request mode.

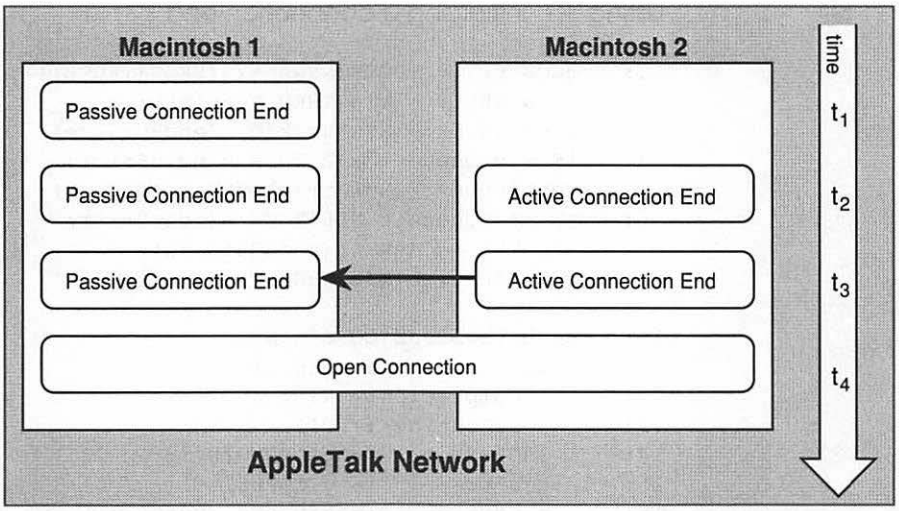


Figure 8-2. Opening a connection using request and passive modes

First one side opens a connection end using the passive mode. This is shown in Figure 8-2 at time t_1. Typically you register an NBP name for the socket used. This allows the other side of the connection to find you using an NBP lookup operation.

At time t_2, a second connection is opened using the request mode. Using the request mode requires you to specify the address of the other connection end you want to connect to. This address is usually found using NBP. Once the request connection is initiated, ADSP attempts to join the two connection ends into an open full connection. This is shown as time t_3.

Finally the connection is declared open, and communication can commence. This is shown as time t_4.

Listings 8-1, 8-2, and 8-3 are code fragments illustrating how to create connection ends using the passive and request modes. Listing 8-1 shows how to initialize a connection end. This is done the same way for both request and passive modes. Listing 8-2 shows how to open a connection end in the passive mode and Listing 8-3 shows how to open a connection end in the request mode.

Listing 8-1. Initializing a connection end

```
 1:   gCCBPtr := TPCCB(NewPtr(SIZEOF(TRCCB)));
 2:   IF gCCBPtr = NIL THEN HandleError;
 3:   sendQueue := NewPtr(kADSPSendBufSize);
 4:   IF sendQueue = NIL THEN HandleError;
 5:   recvQueue := NewPtr(kADSPRecvBufSize);
 6:   IF recvQueue = NIL THEN HandleError;
 7:   attnPtr := NewPtr(attnBufSize);
 8:   IF attnPtr = NIL THEN HandleError;
 9:
10:   WITH theDSPPB DO BEGIN
11:     csCode      := dspInit;
12:     ioCompletion:= NIL;
13:     userRoutine := NIL;
14:     ioCRefNum   := gADSPRefNum;
15:     ccbPtr      := gCCBPtr;
16:     sendQSize   := kADSPSendBufSize;
17:     recvQSize   := kADSPRecvBufSize;
18:     sendQueue   := gSendQueue
19:     recvQueue   := gRecvQueue
20:     attnPtr     := gAttnPtr
21:     localSocket := 0;
22:   END;
23:
24:   IF PBControl(@theDSPPB,kSYNC) <> noErr
25:     THEN HandleError;
```

Lines 1–8 in Listing 8-1 allocate the following four buffers that are required by the connection:

1. The connection control block (CCB)
2. The send buffer
3. The receive buffer
4. The attention buffer

Both the CCB and the attention buffer are of fixed predefined size, but the send and receive buffers are as big or as small as you wish.

Line 11 fills in the **csCode** field with **dspInit** telling the driver that this is an initialization call.

Line 12 fills in the **ioCompletion** field with NIL indicating that no completion routine is to be called.

Line 13 fills in the **userRoutine** field with NIL indicating that no routine should be called when a connection event happens. When this is NIL, you must poll the **userFlags** field of the CCB for connection events.

Line 14 fills in the **ioCRefNum** field. This should have been received previously by opening the .DSP driver.

Line 15 fills in the **ccbPtr** field with a pointer to the CCB that was allocated in line 1.

Lines 16–19 fill in the fields pertaining to the send and receive buffers that were allocated in lines 3 and 5.

Line 20 fills in the **attnPtr** field with a pointer to the attention buffer that was allocated in line 7.

Line 21 indicates that ADSP should select a socket for this connection end. You can specify your own socket by filling in the **localSocket** field with a previously allocated socket.

Line 24 finally makes the call to the .DSP driver. It is done synchronously in this call because the dsp INIT call finished quickly—no network activity is required.

Listing 8-2 illustrates how to open the initialized connection end in the passive mode. It calls the driver asynchronously and later code would then poll for completion to see when the connection is actually opened. Two types of polling are possible here: either check the **ioResult** field of the parameter block to be non-one, or check the **state** field of the CCB to see when it is set to **sOpen**.

Listing 8-2. Opening a connection in passive mode

```
 1:   WITH theDSPPB DO BEGIN
 2:     csCode          := dspOpen;
 3:     ioCompletion    := NIL;
 4:     ioCRefNum       := gADSPRefNum;
 5:     filterAddress   := AddrBlock(0);
 6:     ocMode          := ocPassive;
 7:     ocInterval      := 0;
 8:     ocMaximum       := 0;
 9:   END;
10:
11:   IF PBControl(@theDSPPB,kASYNC) <> noErr
12:     THEN HandleError;
```

Line 2 sets the **csCode** field to be **dspOpen** so the driver will know an open request is being made.

Line 3 sets the **ioCompletion** field to NIL indicating that no completion routine is to be called when the call is finished.

Line 4 fills in the **ioCRefNum** with the reference number of the .DSP driver.

Line 5 sets the **filterAddress** field to be zero. This indicates that open requests should be accepted from any socket address in the network.

Line 6 fills in the **ocMode** field with **ocPassive** telling the driver that this open request should use the passive mode.

Lines 7 and 8 specify that you want to use the default values for the **ocInterval** and **ocMaximum** fields.

Line 11 finally calls the .DSP driver and makes the passive open request. Later on, your code must either poll the **ioResult** field or the **state** field of the CCB to determine when the connection has been opened.

Listing 8-3 illustrates how to open the initialized connection in the request mode. It calls the driver synchronously. Your code may want to call it asynchronously because the opening process can take some time if the remote connection end is busy.

Listing 8-3. Opening session in request mode

```
 1:   WITH theDSPPB DO BEGIN
 2:     csCode        := dspOpen;
 3:     ioCRefNum     := gADSPRefNum;
 4:     remoteAddress := theAddr;
 5:     filterAddress := AddrBlock(0);
 6:     ocMode        := ocRequest;
 7:     ocInterval    := 0;
 8:     ocMaximum     := 0;
 9:   END;
10:
11:   IF PBControl(@theDSPPB,kSYNC) <> noErr
12:     THEN HandleError;
```

Line 2 sets the **csCode** field to be **dspOpen** so the driver will know an open request is being made.

Line 3 fills in the **ioCRefNum** with the reference number of the .DSP driver.

Line 4 sets the **remoteAddress** field to the address of the remote connection end you wish to connect with. This is typically retrieved using an NBP name lookup.

Line 5 sets the **filterAddress** field to be zero. This indicates that you will connect to any socket address in the network. You will normally connect to the socket specified in **remoteAddress**, but when a connection listener is encountered, you will be handed off to another connection that may not be using that same socket.

Line 6 fills in the **ocMode** field with **ocRequest** telling the driver that this open request should use the request mode.

Lines 7 and 8 specify that you want to use the default values for the **ocInterval** and **ocMaximum** fields.

Line 11 finally calls the .DSP driver and makes the open request.

▶ Using a Connection Listener to Open a Session

The other common method for establishing a connection involves a connection listener rather than opening a connection end using the passive mode. When the remote connection end attempts to open a connection using the request mode, thė connection listener hands off the request to a new connection end opened specifically for this purpose. Alternately, the connection listener can deny the connection request. This type of operation is typical of a server environment where a single NBP name is registered to identify the server. This NBP name references the connection listener. Requests come into it from a variety of sources and the connection listener creates connection ends to service the incoming requests. It may have to reject requests when local resources are exhausted.

Figure 8-3 illustrates the type of connection listener scenario just described. It shows a single connection listener on Macintosh 1. A connection request has already come in, resulting in a connection being established with Macintosh 4. Two additional connection requests are just coming in from Macintoshes 2 and 3. The connection listener is creating the passive mode connection ends that will make a full connection with Macintoshes 2 and 3.

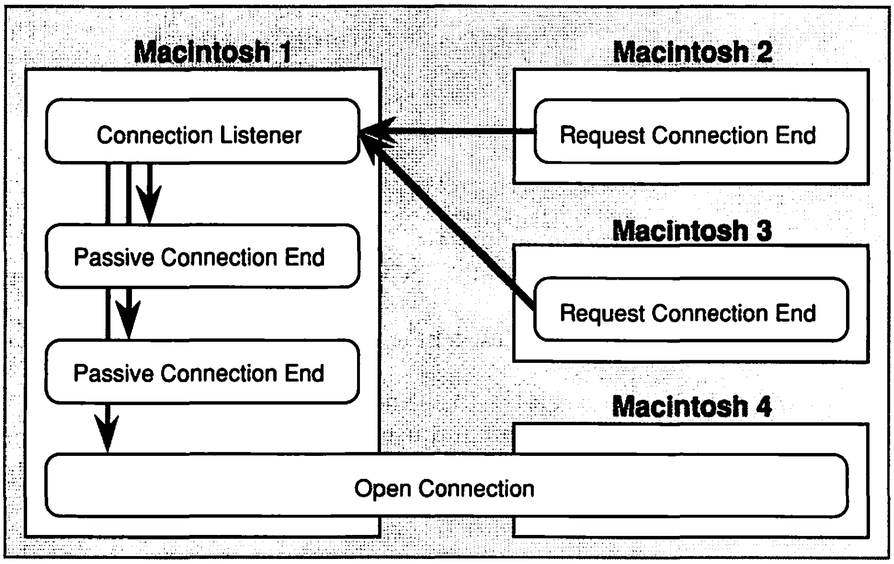


Figure 8-3. Connecting using a connection listener

Listings 8-4, 8-5, and 8-6 show code fragments illustrating how to create and service a connection listener. Listing 8-4 shows how to open a connection listener. Listing 8-5 shows how to open a passive connection end in response to receiving a connection request. Listing 8-6 shows how to deny a connection request.

Listing 8-4 starts out by preparing the parameter block with the values required for the **dspCLInit** call. Line 2 sets the **csCode** field to **dspCLInit** so a connection listener will be initialized when the driver is called.

Listing 8-4. Opening a connection listener

```
 1:   WITH theDSPPB DO BEGIN
 2:     csCode         := dspCLInit;
 3:     ioCRefNum      := gADSPRefNum;
 4:     ccbPtr         := @theCCB;
 5:     localSocket    := 0;
 6:   END;
 7:
 8:   IF PBControl(@theDSPPB,kSYNC) <> noErr
 9:     THEN HandleError;
10:
11:   gCCBRefNum := theDSPPB.ccbRefNum;
12:
13:   WITH theDSPPB DO BEGIN
14:     csCode         := dspCLListen;
15:     ioCRefNum      := gADSPRefNum;
16:     ccbRefNum      := gCCBRefNum;
17:     filterAddress := AddrBlock(0);
18:   END;
19:
20:   IF PBControl(@theDSPPB,kASYNC) <> noErr
21:     THEN HandleError;
```

Line 3 fills in the **icCRefNum** field with the reference number for the .DSP driver.

Line 4 sets the **ccbPtr** field to point to the connection control block set up previously.

Line 5 sets the **localSocket** field to zero, instructing the driver to get any available socket and use it with this connection listener.

Line 8 calls the driver synchronously. You can do it this way since the connection listener initialization function is relatively quick.

Line 11 copies the returned CCB reference number into the global variable **gCCBRefNum** for future reference.

Next, a parameter block is filled in in preparation for calling the **CLListen** call. It starts by setting the **csCode** field to **dspCLListen** to tell the driver that a **CLListen** should be performed.

Lines 15 and 16 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 17 sets the **filterAddress** field to zero indicating that you want to accept connection requests from any address on the network.

Finally line 20 makes the call to the driver asynchronously. This is done so that later your code can poll the **ioResult** field to see if a connection request has come in.

Once a connection request has come in, you usually want to create a new connection end and have it complete the connection request. Listing 8-5 illustrates how to do this.

This code assumes that you are using the same parameter block as was used by the connection listener in Listing 8-4. If you do not use the same parameter block, you need to copy the **remoteCID, remoteAddress, sendSeq, sendWindow,** and **attnSendSeq** fields from the connection listener parameter block to the parameter block used when you make the call to **dspOpen.**

Listing 8-5. Opening a connection using accept mode

```
 1:   WITH theDSPPB^ DO BEGIN
 2:     csCode          := dspOpen;
 3:     ioCRefNum       := gADSPRefNum;
 4:     ccbRefNum       := gCCBRefNum;
 5:     ocMode          := ocAccept;
 6:     ocInterval      := 0;
 7:     ocMaximum       := 0;
 8:   END;
 9:
10:   IF PBControl(@theDSPPB,kSYNC) <> noErr
11:     THEN HandleError;
```

Line 2 sets the **csCode** field to be **dspOpen** so the driver will know an open request is being made.

Lines 3 and 4 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 5 fills in the **ocMode** field with **ocAccept** telling the driver that this open request should use the accept mode.

Lines 6 and 7 specify that you want to use the default values for the **ocInterval** and **ocMaximum** fields.

Line 10 finally calls the .DSP driver and makes the open request.

If rather than granting the open request, you want to deny it, you can use the **CLDeny** call. This will tell the requesting connection end that the connection request has failed. Listing 8-6 shows an example of how this is done. It begins with lines 1 and 2 copying the remote connection ID and remote address from the connection listener's parameter block.

Listing 8-6. Denying a connection request

```
 1:   denyCID      := listeningDSPPB.remoteCID;
 2:   denyAddress  := listeningDSPPB.remoteAddress;
 3:
 4:   WITH theDSPPB DO BEGIN
 5:     csCode        := dspOpen;
 6:     ioCRefNum     := gADSPRefNum;
 7:     ccbRefNum     := gCCBRefNum;
 8:     remoteCID     := denyCID;
 9:     remoteAddress := denyAddress;
10:   END;
11:
12:   IF PBControl(@theDSPPB,kSYNC) <> noErr
13:     THEN HandleError;
```

Line 5 sets the **csCode** field to be **dspCLDeny** so the driver will know a listener deny call is being made.

Lines 6 and 7 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 8 fills in the **remoteCID** field with the remote CID retrieved from the connection listener's parameter block.

Line 9 fills in the **remoteAddress** field with the remote address gotten from the connection listener.

Line 12 finally calls the .DSP driver and completes the open request.

It is also important to remember to call **CLListen** again after you either grant or deny the connection request; that is, if you want to accept further connection requests using the same connection listener.

▶ Sending Data Over a Connection

Once you have set up a connection, the next step is usually to send or receive data over it. This is done using the **dspRead** and **dspWrite** routines. These routines actually only read or write data into or out of the connection's send or receive buffers. The actual transmission or reception of data over the network is done by ADSP independently. This is known as *double buffering*.

Figure 8-4 illustrates double buffering. The three arrows show the three times that data is moved. The first arrow represents the data being moved from the writing program's data area to the connection's send buffer. The second arrow represents the data being transmitted from the local connection end's send buffer over the network into the receive buffer of the remote connection end. The final arrow represents the data being copied from the connection's receive buffer to the reading program's data area.

Because reading and writing ADSP data only copies data into and out of buffers, **dspWrite** and **dspRead** operations complete when the copying is done. The actual transmission of the data may occur some time later.

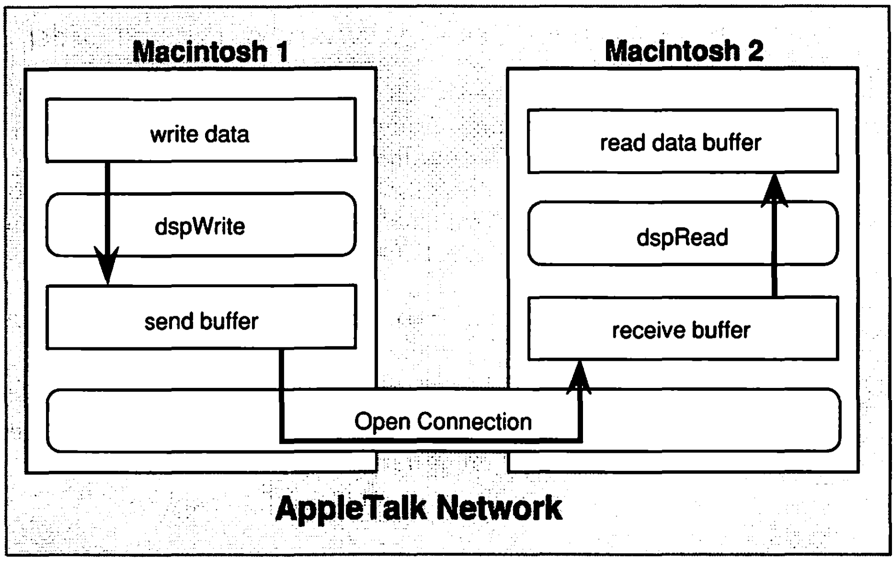


Figure 8-4. Double buffering in ADSP

By the Way ▶

A **dspWrite** operation can involve more data that can fit into the send buffer. ADSP will copy all it can into the send buffer, then wait for the send buffer to empty some of the data. It will then copy more data into the send buffer, repeating this process until it has copied all the data. The **dspWrite** operation *will not complete* until this entire process is done and all the data has been copied into the send buffer. This means that if you call **dspWrite** synchronously, you will be blocked until all the copying is done. This can involve long periods of time. Of course if you call it asynchronously you can continue processing and use a completion routine or polling to tell when the **dspWrite** operation has finished.

▶ How ADSP Decides When to Transmit Data

ADSP determines when to transmit the data in the send queue based on a number of factors. ADSP will not transmit the data in the send buffer if the receive buffer on the other end is full. ADSP always waits until there is free space in the receive buffer before attempting to send any data. If this condition is satisfied, ADSP will send data when any of the following four conditions is satisfied:

- The send buffer contains as many bytes or more than the blocking factor.
- The send timer expires with data in the send buffer.
- An acknowledgment packet must be sent to the remote connection end and there is data in the send buffer.
- The flush flag was set in a call to dspWrite.

The first three conditions will happen without your intervention, while the last condition, using the **flush** flag in a **dspWrite**, is how you can force the transmission of data.

▶ The Structure of the Data Stream

ADSP treats the data it handles as a continuous stream of bytes. At times this is precisely what is needed. For example, when emulating the behavior of a serial line you would send and receive a simple stream of ASCII data across a connection.

At other times it is useful to impose some structure on the data sent over a connection. For this purpose, ADSP provides a mechanism for grouping data into messages. When these messages are sent, they are terminated by a logical **end-of-message**. This logical **end-of-message** forces a read on the receiver end of the connection to finish even if the requested number of bytes has not been copied. Also, note that you never see this logical **end-of-message** as data; you just observe its effect.

For example, say you wish to send a series of Pascal strings across a connection. These strings vary in size and you only want to send the actual data; you don't want to pad them all out to a fixed 256 bytes in length. Say your first two strings are "Hello" and "Goodbye." You would send the five characters of the first string using **dspWrite** with the **end-of-message** flag set, then send the seven characters of the second string with the **end-of-message** flag set.

On the receiver side, your program would have made a **dspRead** asking for 256 bytes of data. It completes after only receiving the "Hello" data, but the **actCount** field would contain five rather than 256, telling you that only five bytes had actually been received. Likewise, a second read asking for 256 bytes would complete after receiving the "Goodbye" data and the **actCount** field would tell you that seven bytes had actually been received.

By the Way ▶

Because of the way the logical **end-of-message** is implemented, every time you send a logical **end-of-message**, ADSP sends a separate DDP packet. This can result in a large number of very small packets if you send a small amout of data between logical **end-of-message** flags. This can have a detrimental impact on overall throughput.

▶ Using dspRead and dspWrite

Listing 8-7 shows a routine that sends a single string using ADSP. It sets the **end-of-message** and **flush** flags so that the data will be sent as one message right away.

Listing 8-7. Sending a string using dspWrite

```
 1: PROCEDURE SendString(dataString : Str255);
 2: BEGIN
 3:   WITH theDSPPB DO BEGIN
 4:     csCode         := dspWrite;
 5:     ioCRefNum      := gADSPRefNum;
 6:     ccbRefNum      := gCCBRefNum;
 7:     reqCount       := LENGTH(dataString)+1;
 8:     dataPtr        := @dataString;
 9:     eom            := 1;
10:     flush          := 1;
11:   END;
12:
13:   IF PBControl(@theDSPPB,kSYNC) <> noErr
14:     THEN HandleError;
15: END;
```

Line 4 sets the **csCode** field to be **dspWrite** so the driver will know a write call is being made.

Lines 5 and 6 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 7 sets the **reqCount** field to be the length of the string that is being sent plus 1 for the Pascal string's length byte.

Line 8 sets the **dataPtr** field to point to the string data.

Lines 9 and 10 set the flags that tell the driver to send a logical end-of-message after this message and to send it as soon as possible.

Line 13 finally calls the .DSP driver and does a write operation.

Listing 8-8 shows a routine that will read the strings sent by the routine in Listing 8-7.

Listing 8-8. Reading a string using dspRead

```
 1: PROCEDURE GetString(VAR dataString : Str255;
 2:                     VAR bytesRead : integer);
 3: BEGIN
 4:   WITH theDSPPB DO BEGIN
 5:     csCode         := dspWrite;
 6:     ioCRefNum      := gADSPRefNum;
 7:     ccbRefNum      := gCCBRefNum;
 8:     reqCount       := 256;
 9:     dataPtr        := @dataString;
10:   END;
11:
```

```
12:     IF PBControl(@theDSPPB,kSYNC) <> noErr
13:       THEN HandleError;
14:
15:     bytesRead := theDSPPB.actCount;
16:  END;
```

Line 5 sets the **csCode** field to be **dspWrite** so the driver will know a write call is being made.

Lines 6 and 7 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 8 sets the **reqCount** field to 256, the maximum length of a string.

Line 9 sets the **dataPtr** field to point to the string data.

Line 13 calls the .DSP driver and does a read operation.

Line 15 copies the **actCount** field into the return parameter **bytesRead,** telling the caller how many bytes were actually read.

▶ Using dspStatus

ADSP provides a way to get useful information about an open connection. The **dspStatus** call lets you check the status of the send and receive buffers and also returns a pointer to the connection's CCB.

The information returned by **dspStatus** lets you see two things about the send and receive buffers: how many bytes are free, and how many bytes are used. This has a number of useful applications.

Suppose, for example, that you would like to use the synchronous forms of **dspRead** and **dspWrite** so that you don't have to poll for completion. If you attempt to write more data into the send buffer than there is space for, your write call will hang until ADSP can make room for your data. By using **dspStatus,** you can peek into the send buffer and see if there is enough free space to hold the number of bytes you want to send. If there isn't, you can wait until there is.

Likewise, say you want to read 20 bytes of data. You can use **dspStatus** to see how many bytes of data are waiting there for you. Once you know the data is there, you can safely do a synchronous read and know that you won't be blocked.

▶ Attention Messages

Another feature of ADSP is the ability to send attention messages without disrupting the normal message flow. This is known as *out-of-band signaling.*

Attention messages can be useful in a wide variety of circumstances. For example, when sending unstructured data across the connection, control information can be exchanged using attention messages. Another example would be sending status information while doing a large data transfer.

An ADSP attention message consists of a 16-bit attention code along with up to 570 bytes of additional data. The attention code must be in the range $0000 through $EFFF. The attention codes in the range $F000 through $FFFF are reserved for use by ADSP itself.

When an attention messsage is issued, ADSP sends it to the remote connection end before sending any more data. This assures that the attention message is delivered as quickly as possible.

When an attention message is received by the remote connection end, ADSP sets the **eAttention** bit (bit 5) in the **userFlags** field of the remote connection end's connection control block. It will also call the routine specified in the **userRoutine** field of the **dspInit** call if one was specified. This allows two ways to detect attention messages: either poll the **eAttention** bit of the **userFlags** or have a completion routine called.

Once the attention message has been received, you must set the **userFlags** to zero. This allows another attention message, or other unsolicited connection event, to occur. Failure to clear the **userFlags** will result in your connection hanging.

Listing 8-9 shows an example of how to send an attention message.

Listing 8-9. Sending an attention message

```
 1:   WITH theDSPPB DO BEGIN
 2:     csCode          := dspAttention;
 3:     ioCRefNum       := gADSPRefNum;
 4:     ccbRefNum       := gCCBRefNum;
 5:     attnCode        := kWakeup;
 6:     attnData        := @WakeUpMessage;
 7:     attnSize        := SIZEOF(WakeUpMessageRec);
 8:   END;
 9:
10:   IF PBControl(@theDSPPB,kSYNC) <> noErr
11:     THEN HandleError;
```

Line 1 sets the **csCode** field to be **dspAttention** so the driver will know an attention message is being sent.

Lines 3 and 4 set the **ioCRefNum** and **ccbRefNum** fields to the appropriate reference numbers.

Line 5 sets the **attnCode** field to be **kWakeUp**, a constant known to both sides of the connection.

Lines 6 and 7 set the **attnData** field to point to the wakeup message buffer and **attnSize** field to the size of this data.

Line 10 finally calls the .DSP driver and sends the attention message.

▶ Detailed Descriptions of Important ADSP Routines

The following section describes each important ADSP routine. It shows the routine's prototype, lists all parameters or parameter block fields, and lists error codes. Each parameter or parameter block field and error code is then described in detail.

▶ dspInit

dspInit is used to create and initialize a connection end. It must be called before the connection is opened. The **dspInit** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          – always dspInit
  →    ioCompletion    – address of completion routine
  ←    ioResult        – result of operation
  →    ioCRefNum       – reference number of .DSP driver
  ←    ccbRefNum       – reference number of connection
                         control block
  →    ccbPtr          – pointer to the connection control
                         block
  →    userRoutine     – address of connection event routine
  →    sendQSize       – size of the send queue
  →    sendQueue       – pointer to the send queue
  →    recvQSize       – size of the receive queue
  →    recvQueue       – pointer to the receive queue
  →    attnPtr         – pointer to attention buffer
  ↔    localSocket     – socket number for the connection end

Errors Returned:

 noErr (0)                 No error.
 ddpSktErr (-91)           Error when opening DDP socket.
 errDSPQueueSize (-1274)   Send or receive queue is too
                           small.
```

csCode always contains **dspInit**, that is, the constant 255.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum returns the reference number assigned to the connection control block for this connection end.

ccbPtr contains the pointer to a connection control block that will be used by the connection end.

userRoutine contains the address of the **userRoutine** routine called when an unsolicited connection event happens on this connection end. This should be set to NIL if no **userRoutine** is desired. This routine is called under the same conditions as a completion routine (at interrupt level) and must follow the same rules as a completion routine.

sendQSize contains the size of the send buffer. The minimum size for a send buffer is 100 bytes, which is found in the constant **minDSPQueueSize**.

sendQueue contains a pointer to the send buffer.

recvQSize contains the size of the receive buffer. The minimum size for a receive buffer is 100 bytes, which is found in the constant **minDSPQueueSize**.

recvQueue contains a pointer to the receive buffer.

attnPtr contains a pointer to the attention buffer. This buffer must be 570 bytes in size, which is found in the constant **attnBufSize**.

localSocket contains the socket number that this connection end should use. If it is set to zero, ADSP will assign a new socket to the connection end and return the socket number in this field.

noErr is returned when the trap completes normally.

ddpSktErr is returned when DDP encounters an error opening the socket.

errDSPQueueSize is returned when ADSP determines that the supplied send or receive buffers are smaller than the minimum required (100 bytes).

▶ dspOptions

dspOptions is used to set a number of parameters pertaining to a connection end. The **dspOptions** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspOptions
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    sendBlocking    - size at which data is sent
  →    badSeqMax       - threshold for retransmission
                         request
  →    useCheckSum     - indicates use of DDP checksums

Errors Returned:

  noErr (0)            No error.
  errRefNum (-1280)    Unknown connection reference number.
```

csCode always contains **dspOptions**, that is, the constant 243.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

sendBlocking contains the new value for the send blocking factor. It specifies the maximum number of bytes that should accumulate in the send buffer before an attempt is made to transmit the bytes. If this is set to zero, no change is made to the blocking factor. Other valid values range from 1 to 572—the maximum number of bytes found in a DDP packet. A default value of 16 is set by **dspInit**.

badSeqMax contains the new value for the bad sequence factor. It specifies the maximum number of out-of-sequence packets that will be received before ADSP will make a special request for the missing packets. If **badSeqMax** is set to zero, no change is made to the bad sequence factor. Other valid values range from 1 to 255. A default value of 3 is set by **dspInit**.

useCheckSum contains a flag specifying whether DDP should use checksums for all packets transmitted. A value of 1 means to use DDP checksums; a value of 0 means not to. A default of not using DDP checksums is set by **dspInit**.

noErr is returned when the trap completes normally.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspOpen

dspOpen is used to open a connection end. Because the four modes available for opening connection ends use difference fields in the parameter block, each mode is listed separately below.

dspOpen in Passive Mode

dspOpen in passive mode is used to open a connection end that can later be connected to by another connection end using request mode. The **dspOpen** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspOpen
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    ocMode          - always ocPassive
  →    ocInterval      - interval between open request
                        retransmissions
  →    ocMaximum       - max number of open request
                        retransmissions
  ←    localCID        - local connection end ID
  ←    remoteCID       - remote connection end ID
  ←    remoteAddress   - address of remote connection end
  →    filterAddress   - addresses that are acceptable for
                         connections
  ←    sendSeq         - sequence # of the first byte sent
  ←    sendWindow      - sequence # of the last remote byte
  ←    attnSendSeq     - sequence # of the next attention to
                         be sent
```

```
Errors  Returned:

  noErr  (0)                  No  error.
  errOpenDenied  (-1273)      Open  request  denied.
  errOpening  (-1277)         Open  request  failed.
  errState  (-1278)           Connection  end  is  not  closed.
  errAborted  (-1279)         Request  aborted  by  either
                              dspRemove  or  dspClose.
  errRefNum  (-1280)          Connection  reference  number  was  bad.
```

csCode always contains **dspOpen**, that is, the constant 253.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

ocMode always contains **ocPassive**, that is, the constant 2.

ocInterval and **ocMaximum** determine the retry behavior of the open request. **ocMaximum** tells how many retries should be attempted and **ocInterval** tells how long to wait between retries in 10-tick units.

localCID returns the connection ID of the local connection end.

remoteCID returns the connection ID of the remote connection end.

remoteAddress returns the socket address of the remote connection end.

filterAddress contains the address from which connection requests will be accepted. A zero value in the network number, node ID, or socket number indicates that any value for those fields will be accepted.

sendSeq, **sendWindow**, and **attnSendSeq** return synchronization information for the connection. When using the passive mode, this is returned for informational purposes only.

noErr is returned when the trap completes normally.

errOpenDenied is returned when the open request has been denied.

errOpening is returned when the open request fails.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

dspOpen in Request Mode

dspOpen in request mode is used to open a connection end that will complete the connection with the specified remote connection end. The **dspOpen** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspOpen
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    ocMode          - always ocRequest
  →    ocInterval      - interval between open request
                         retransmissions
  →    ocMaximum       - max number of open request
                         retransmissions
  ←    localCID        - local connection end ID
  ←    remoteCID       - remote connection end ID
  →    remoteAddress   - address of remote connection end
  →    filterAddress   - addressess that are acceptable for
                         connections
  ←    sendSeq         - sequence # of the first byte sent
  ←    sendWindow      - sequence # of the last remote byte
  ←    attnSendSeq     - sequence # of the next attention to
                         be sent

Errors Returned:

  noErr (0)              No error.
  errOpenDenied (-1273)  Open request denied.
  errOpening (-1277)     Open request failed.
  errState (-1278)       Connection end is not closed.
  errAborted (-1279)     Request aborted by either dspRemove
                         or dspClose.
  errRefNum (-1280)      Connection reference number was bad.
```

csCode always contains **dspOpen**, that is, the constant 253.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

ocMode always contains **ocRequest**, that is, the constant 1.

ocInterval and **ocMaximum** determine the retry behavior of the open request. **ocMaximum** tells how many retries should be attempted and **ocInterval** tells how long to wait between retries in 10-tick units.

localCID returns the connection ID of the local connection end.

remoteCID returns the connection ID of the remote connection end.

remoteAddress contains the socket address of the remote connection end or connection listener that you wish to communicate with. This contains the socket address of the remote connection end you finally establish an open connection with. This may be a different address than you specified because a connection listener can connect you with a connection end at another socket.

filterAddress contains the address from which connection requests will be accepted. A zero value in any of either the network number, node ID, or socket number indicates that any value for those fields will be accepted. You can use this to restrict the range of socket addresses that a remote connection listener can connect you with.

sendSeq, sendWindow, and **attnSendSeq** return synchronization information for the connection. When using the passive mode, this is returned for informational purposes only.

noErr is returned when the trap completes normally.

errOpenDenied is returned when the open request has been denied.

errOpening is returned when the open request fails.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

dspOpen in Accept Mode

dspOpen in accept mode is used by a connection listener to complete a connection request.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async: BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspOpen
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    ocMode          - always ocAccept
  →    ocInterval      - interval between open request
                         retransmissions
  →    ocMaximum       -  max number of open request
                         retransmissions
  ←    localCID        - local connection end ID
  →    remoteCID       - remote connection end ID
  →    remoteAddress   - address of remote connection end
  ←    sendSeq         - sequence # of the first byte sent
  ←    sendWindow      - sequence # of the last remote byte
  ←    attnSendSeq     - sequence # of the next attention to
                         be sent

Errors Returned:

  noErr (0)               No error.
  errOpenDenied (-1273)   Open request denied.
  errOpening (-1277)      Open request failed.
  errState (-1278)        Connection end is not closed.
  errAborted (-1279)      Request aborted by either
                          dspRemove or dspClose.
  errRefNum (-1280)       Connection reference number was bad.
```

csCode always contains **dspOpen**, that is, the constant 253.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

ocMode always contains **ocAccept**, that is, the constant 3.

ocInterval and **ocMaximum** determine the retry behavior of the open request. **ocMaximum** tells how many retries should be attempted and **ocInterval** tells how long to wait between retries in 10-tick units.

localCID returns the connection ID of the local connection end.

remoteCID contains the connection ID of the remote connection end; this is received from the connection listener.

remoteAddress contains the socket address of the remote connection end; this is gotten from the connection listener.

sendSeq, **sendWindow**, and **attnSendSeq** contain synchronization information; this is retrieved from the connection listener.

noErr is returned when the trap completes normally.

errOpenDenied is returned when the open request has been denied.

errOpening is returned when the open request fails.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

dspOpen in Establish Mode

dspOpen in establish mode is used to open a connection end when the entire setup process is handled by you. The **dspOpen** routine follows.

```
FUNCTIONPBControl(theDSPPBPtr:DSPPBPtr; async: BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode           - always dspOpen
  →    ioCompletion     - address of completion routine
  ←    ioResult         - result of operation
  →    ioCRefNum        - reference number of .DSP driver
  →    ccbRefNum        - reference number of connection
                          control block
  →    ocMode           - always ocEstablish
  →    remoteCID        - remote connection end ID
  →    remoteAddress    - address of remote connection end
  →    sendSeq          - sequence # of the first byte sent
  →    sendWindow       - sequence # of the last remote byte
  →    recvSeq          - sequence # of the first byte to
                          receive
  →    attnSendSeq      - sequence # of the next attention to
                          be sent
  →    attnRecvSeq      - sequence # of the next attention to
                          receive
Errors Returned:

  noErr (0)              No error.
  errOpenDenied (-1273)  Open request denied.
  errOpening (-1277)     Open request failed.
  errState (-1278)       Connection end is not closed.
  errAborted (-1279)     Request aborted by either dspRemove
                         or dspClose.
  errRefNum (-1280)      Connection reference number was bad.
```

csCode always contains **dspOpen**, that is, the constant 253.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

ocMode always contains **ocEstablish**, that is, the constant 4.

remoteCID contains the connection ID of the remote connection end.

remoteAddress contains the socket address of the remote connection end.

sendSeq, **sendWindow**, **recvSeq**, **attnSendSeq**, and **attnRecvSeq** contain synchronization information for the connection.

noErr is returned when the trap completes normally.

errOpenDenied is returned when the open request has been denied.

errOpening is returned when the open request fails.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspNewCID

dspNewCID is used to create a connection ID for your use in setting up your own connection. This should only be used with the establish mode of the open connection call. The **dspNewCID** routine follows.

```
FUNCTION  PBControl(theDSPPBPtr:DSPPBPtr;  async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

   →    csCode          - always dspNewCID
   →    ioCompletion    - address of completion routine
   ←    ioResult        - result of operation
   →    ioCRefNum       - reference number of .DSP driver
   →    ccbRefNum       - reference number of connection
                          control block
   →    theNewCID       - new connection ID

Errors Returned:

   noErr  (0)            No error.
   errState  (-1278)     Connection end is not closed.
   errRefNum  (-1280)    Connection reference number was bad.
```

csCode always contains **dspNewCID**, that is, the constant 241.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

theNewCID contains the new connection ID for use when opening a connection using the establish mode.

noErr is returned when the trap completes normally.

errState is returned when the connection is not in the closed state and an open request is made.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspClose

dspClose is used to close a connection end. After closing the connection end still exists and can be opened again later. You should use **dspRemove** if you want to destroy the connection end and release its resources. The **dspClose** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode           - always dspClose
  →    ioCompletion     - address of completion routine
  ←    ioResult         - result of operation
  →    ioCRefNum        - reference number of .DSP driver
  →    ccbRefNum        - reference number of connection
                          control block
  →    abort            - abort outstanding send requests

Errors Returned:

  noErr (0)           No error.
  errState (-1278)    Connection end is not closed.
  errRefNum (-1280)   Connection reference number was bad.
```

csCode always contains **dspClose**, that is, the constant 252.
ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.
ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.
ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**
ccbRefNum contains the reference number for the connection control block.
abort contains a flag indicating if outstanding **dspWrite** or **dspAttention** calls should be aborted or not. Set the flag to 1 to abort the **dspWrite** and **dspAttention** calls or 0 to allow them to complete before closing the connection.
noErr is returned when the trap completes normally.
errState is returned when the connection is not open.
errRefNum is returned when the supplied connection reference number is illegal.

▶ dspCLInit

dspCLInit initializes a connection listener. You should call **dspCLListen** to begin receiving a connection request on this listener. The **dspCLInit** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspCLInit
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    ccbPtr          - pointer to the connection control
                         block
  ↔    localSocket     - socket number for the connection
                         listener

Errors Returned:

  noErr (0)            No error.
  ddpSktErr (-91)      Error when opening DDP socket.
```

csCode always contains **dspCLInit**, that is, the constant 251.
ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.
ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.
ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.
ccbRefNum contains the reference number for the connection control block.
ccbPtr contains the pointer to a connection control block that will be used by the connection listener.
localSocket contains the socket number that this connection listener should use. If it is set to zero, ADSP will assign a new socket to the connection listener and return the socket number in this field.
noErr is returned when the trap completes normally.
ddpSktErr is returned when DDP encounters an error opening the socket.

▶ dspCLListen

dspCLListen initiates the connection to listen for connection requests. It is almost always called asynchronously. The **dspCLListen** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →     csCode           - always dspCLListen
  →     ioCompletion     - address of completion routine
  ←     ioResult         - result of operation
  →     ioCRefNum        - reference number of .DSP driver
  →     ccbRefNum        - reference number of connection
                           control block
  ←     remoteCID        - remote connection end ID
  ←     remoteAddress    - address of remote connection end
  →     filterAddress    - addresses that are acceptable for
                           connections
  ←     sendSeq          - sequence # of the first byte sent
  ←     sendWindow       - sequence # of the last remote byte
  ←     attnSendSeq      - sequence # of the next attention to
                           be sent
```

```
Errors Returned:
  no err (0)              No error.
  errState (-1278)        Connection end is not closed.
  errAborted (-1279)      Request aborted by dspCLRemove.
  errRefNum (-1280)       Connection reference number was bad.
```

csCode always contains **dspCLListen**, that is, the constant 249.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

remoteCID returns the connection ID of the remote connection end. This should be passed along to the connection open call or **dspCLDeny**.

remoteAddress returns the socket address of the remote connection end. This should be passed along to the connection open call or **dspCLDeny**.

filterAddress contains the address from which connection requests will be accepted. A zero value in any of either the network number, node ID, or socket number indicates that any value for those fields will be accepted.

sendSeq, **sendWindow**, and **attnSendSeq** return synchronization information for the connection. This should be passed along to the connection open call.

noErr is returned when the trap completes normally.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the listen request is aborted by the **dspCLRemove** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspCLDeny

dspCLDeny is used to deny a connection request. Remember to call **dspCLListener** again if you wish to continue to listen for further connection requests. The **dspCLDeny** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspCLDeny
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    remoteCID       - remote connection end ID
  →    remoteAddress   - address of remote connection end

Errors Returned:

  noErr (0)            No error.
  errState (-1278)     Connection end is not closed.
  errAborted (-1279)   Request aborted by either dspCLRemove
                      or dspClose.
  errRefNum (-1280)    Connection reference number was bad.
```

csCode always contains **dspCLDeny**, that is, the constant 248.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

remoteCID contains the connection ID of the remote connection end. This should be retrieved from **dspCLListen**.

remoteAddress contains the socket address of the remote connection end. This should be retrieved from **dspCLListen**.

noErr is returned when the trap completes normally.

errState is returned when the connection is not in the closed state and an open request is made.

errAborted is returned when the deny request is aborted by either the **dspCLRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspCLRemove

dspCLRemove is used to close a connection listener. You should release the memory used by the CCB if you expect to open the connection listener again later. The **dspCLRemove** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →     csCode          – always dspCLRemove
  →     ioCompletion    – address of completion routine
  ←     ioResult        – result of operation
  →     ioCRefNum       – reference number of .DSP driver
  →     ccbRefNum       – reference number of connection
                          control block
  →     abort           – abort outstanding listen and deny
                          requests

Errors Returned:

  noErr (0)            No error.
  errRefNum (-1280)    Connection reference number was bad.
```

csCode always contains **dspCLRemove**, that is, the constant 250.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

abort contains a flag indicating if outstanding **dspCLListen** or **dspCLDeny** calls should be aborted or not. Set this to 1 to abort them or to 0 to allow them to complete sending any data before closing the connection.

noErr is returned when the trap completes normally.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspStatus

dspStatus is used to get information about an open connection. It returns information about the send and receive queues and the current CID. The **dspStatus** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspStatus
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  ←    statusCCB       - pointer to the connection's CCB
  ←    sendQPending    - amount of data to be sent or
                         acknowledged
  ←    sendQFree       - amount of send buffer space free
  ←    recvQPending    - amount of data to be read from
                         receive buffer
  ←    recvQFree       - amount of receive buffer space free

Errors Returned:

  noErr (0)            No error.
  errRefNum (-1280)    Connection reference number was bad.
```

csCode always contains **dspStatus**, that is, the constant 247.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

statusCCB returns the connection control block of the specified connection.

sendQPending returns the number of bytes that are in the send queue waiting to be sent. Included in the count is each logical end-of-message indicator (counted as 1 byte). Some of this data may have already been sent to the remote connection end, but not acknowledged.

sendQFree returns the number of bytes free in the send queue.

recvQPending returns the number of bytes that are in the receive queue waiting to be read. Included in the count is each logical end-of-message indicator (counted as 1 byte).

recvQFree returns the number of bytes free in the receive queue.

noErr is returned when the trap completes normally.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspWrite

dspWrite is used to send data across an ADSP connection. The **dspWrite** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          - always dspWrite
  →    ioCompletion    - address of completion routine
  ←    ioResult        - result of operation
  →    ioCRefNum       - reference number of .DSP driver
  →    ccbRefNum       - reference number of connection
                         control block
  →    reqCount        - number of bytes requested to send
  ←    actCount        - number of bytes actually sent
  →    dataPtr         - pointer to data buffer
  →    eom             - flag indicating end-of-message
                         should be sent
  →    flush           - flag indicating connection
                         should be flushed

Errors Returned:

  noErr (0)            No error.
  errState (-1278)     Connection end is not open.
  errAborted (-1279)   Request aborted by either dspRemove
                       or dspClose.
  errRefNum (-1280)    Connection reference number was bad.
```

csCode always contains **dspWrite**, that is, the constant 245.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

reqCount contains the number of bytes to send.

actCount returns the number of bytes actually sent. Fewer bytes are sent than requested if the **dspWrite** operation aborts.

dataPtr contains the pointer to the data that should be sent.

eom contains a flag indicating if a logical **end-of-message** should be sent after the data. A value of 1 indicates an **end-of-message** should be sent; a value of 0 indicates that an **end-of-message** shouldn't be sent.

flush contains a flag indicating if the data should be forced to be immediately sent to the remote connection. A value of 1 indicates the data should be sent immediately; a value of 0 indicates that it shouldn't.

noErr is returned when the trap completes normally.

errState is returned when the connection is not in the open state.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspRead

dspRead is used to read data from an ADSP connection. The **dspRead** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          – always dspRead
  →    ioCompletion    – address of completion routine
  ←    ioResult        – result of operation
  →    ioCRefNum       – reference number of .DSP driver
  →    ccbRefNum       – reference number of connection
                         control block
  →    reqCount        – number of bytes requested to send
  ←    actCount        – number of bytes actually sent
  →    dataPtr         – pointer to data buffer
  ←    eom             – flag indicating end-of-message
                         should be sent

Errors Returned:

  noErr (0)            No error.
  errFwdReset )-1275)  Read was terminated by a forward
                       reset.
  errState (-1278)     Connection end is not open.
  errAborted (-1279)   Request aborted by either dspRemove
                       or dspClose.
  errRefNum (-1280)    Connection reference number was bad.
```

csCode always contains **dspRead**, that is, the constant 246.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

reqCount contains the number of bytes you want to read.

actCount returns the number of bytes you actually read.

dataPtr contains the pointer to the data buffer where you should put the read data.

eom returns a flag indicating if the last byte read was the logical end-of-message indicator. When this is TRUE, your **actCount** will often not equal your **reqCount**.

noErr is returned when the trap completes normally.
errFwdReset is returned when the read operation is terminated by a forward reset coming from the remote connection end.
errState is returned when the connection is not in the open state.
errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.
errRefNum is returned when the supplied connection reference number is illegal.

▶ dspAttention

You use **dspAttention** to send an attention message. The **dspAttention** routine follows.

```
FUNCTION PBControl(theDSPPBPtr:DSPPBPtr; async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          – always dspAttention
  →    ioCompletion    – address of completion routine
  ←    ioResult        – result of operation
  →    ioCRefNum       – reference number of .DSP driver
  →    ccbRefNum       – reference number of connection
                         control block
  →    attnCode        – attention code
  →    attnSize        – size of attention message
  →    attnData        – pointer to attention message data

Errors Returned:

  noErr (0)             No error.
  errAttention (-1276)  Attention message is too large.
  errState (-1278)      Connection end is not open.
  errAborted (-1279)    Request aborted by either dspRemove
                        or dspClose.
  errRefNum (-1280)     Connection reference number was bad.
```

csCode always contains **dspAttention**, that is, the constant 244.
ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.
ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen.**

ccbRefNum contains the reference number for the connection control block.

attnCode contains the two-byte attention code. This must be in the range $0000 through $EFFF.

attnSize contains the number of bytes in the attention message.

attnData contains the pointer to the attention message data.

noErr is returned when the trap completes normally.

errAttention is returned when the specified attention message is larger than 570 bytes.

errState is returned when the connection is not in the open state.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ dspReset

Use **dspReset** to clear all data sent from the connection and to resynchronize the connection. When this call is issued, all data not yet received by the remote connection end is lost. The **dspReset** routine follows.

```
FUNCTION  PBControl(theDSPPBPtr:DSPPBPtr;  async:BOOLEAN):OSErr;

Using the following fields in the ADSP Parameter Block:

  →    csCode          – always dspReset
  →    ioCompletion    – address of completion routine
  ←    ioResult        – result of operation
  →    ioCRefNum       – reference number of .DSP driver
  →    ccbRefNum       – reference number of connection
                         control block
Errors Returned:

  noErr  (0)               No error.
  errAttention  (-1276)    Attention message is too large.
  errState  (-1278)        Connection end is not open.
  errAborted  (-1279)      Request aborted by either dspRemove
                           or dspClose.
  errRefNum  (-1280)       Connection reference number was bad.
```

csCode always contains **dspReset**, that is, the constant 242.

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

ioCRefNum contains the reference number for the .DSP driver. This is usually retrieved by opening it with **PBOpen**.

ccbRefNum contains the reference number for the connection control block.

noErr is returned when the trap completes normally.

errAttention is returned when too much data is specified for the attention message.

errState is returned when the connection is not in the open state.

errAborted is returned when the open request is aborted by either the **dspRemove** or the **dspClose** call.

errRefNum is returned when the supplied connection reference number is illegal.

▶ Summary

This chapter covered the AppleTalk Data Stream Protocol. It illustrated what connections are and how they can be created using a variety of techniques. This chapter also covered how to read and write data over ADSP and how to send and receive attention messages.

Chapter 9 discusses miscellaneous AppleTalk interfaces such as drivers, SelfSend, and the Chooser. The last section of Chapter 9 is about configuring a Chooser interface. This section also explains the 'STR', 'GNRL', 'nrct', 'LDEF', and 'PACK' resources.

9 ▶ Miscellaneous AppleTalk Interfaces

When programming AppleTalk on the Macintosh, there are a number of lesser issues that come up beyond using the main protocols. These issues range from using the Chooser as a user interface to detecting when AppleTalk is turned on or off. This chapter deals with these issues, taking each one in turn, and providing you with insight into how to deal with them.

▶ Opening the AppleTalk Drivers

In the old days, initializing AppleTalk for your program involved a variety of calls and reading low memory. This complicated matters and led to some questionable programming practices, such as accessing low memory directly, which should be avoided at all costs since Apple has said this may break in future system software. Modern AppleTalk programs (those using System 6 or beyond) don't have to fool with those older techniques. You should not use the older techniques in order to be compatible with future system software and alternate operating environments such as A/UX.

▶ Opening the .MPP Driver

In order to access the basic AppleTalk protocols, including Name Binding Protocol (NBP) and AppleTalk Transaction Protocol (ATP), among others, you should open the driver called .MPP. Opening this driver checks a variety of things that you were required to do for yourself in the past. It also opens ATP, which used to require a separate operation. This simplifies the AppleTalk opening process significantly. Listing 9-1 shows how this is done. It defines a routine for opening the .MPP driver. This routine returns TRUE if it succeeds and FALSE if it fails.

Listing 9-1. Opening the .MPP driver

```
FUNCTION myOpenMPP : BOOLEAN;
VAR
  refNum : integer;
BEGIN
  myOpenMPP :=
      OpenDriver('.MPP',refNum) = noErr;
END;
```

The major reason for the call to **OpenDriver** to fail is that the user has turned off AppleTalk. A reasonable behavior in many circumstances is to put up a dialog telling the user that AppleTalk is not available, asking them to please turn it back on.

▶ Opening the .XPP Driver

The .MPP driver does not contain all of the AppleTalk protocols. An additional driver must be opened in order to access AppleTalk Echo Protocol (AEP), AppleTalk Session Protocol(ASP), and AppleTalk Filing Protocol(AFP). These additional protocols can be found in the .XPP driver.

Opening the .XPP driver is done in a similar way to opening the .MPP driver. The .XPP driver should be opened after you have opened the .MPP driver. Listing 9-2 shows how this is done. It defines a routine that attempts to open the .XPP driver and returns TRUE if it succeeds, or FALSE if it doesn't.

Listing 9-2. Opening the .XPP driver.

```
FUNCTION myOpenXPP : BOOLEAN;
VAR
  refNum : integer;
BEGIN
  myOpenXPP :=
      OpenDriver('.XPP',refNum) = noErr;
END;
```

▶ Opening the .DSP driver

The AppleTalk Data Stream Protocol (ADSP) arrived on the scene after the .XPP driver. So Apple put ADSP into its own driver that must be opened separately from the .MPP driver and the .XPP driver—this is the .DSP driver.

Opening the .DSP driver is done in a similar way to opening the .MPP and .XPP drivers. The .DSP driver should be opened after you have opened the .MPP driver. Listing 9-3 shows how this is done. It defines a routine that attempts to open the .DSP driver and returns TRUE if this open succeeds, or FALSE if it doesn't.

Listing 9-3. Opening the .DSP driver

```
FUNCTION myOpenDSP : BOOLEAN;
VAR
  refNum : integer;
BEGIN
  myOpenDSP :=
      OpenDriver('.DSP',refNum) = noErr;
END;
```

▶ PSelfSend

In the early days of AppleTalk, Apple decided that networking was for communicating across the network and not for talking to yourself on the same machine. This made some sense because MultiFinder hadn't arrived on the scene and it didn't really make too much sense to have a program use AppleTalk to talk to itself rather than calling its own routines directly. But times changed—MultiFinder let users run more than one program at a time on a single machine. Programmers discovered INITs and this led to a proliferation of small programs all running along with the applications. Finally someone in the AppleTalk group decided that having a Macintosh talk to itself wasn't such a silly idea after all.

The call **PSelfSend** was born. This handy little call allows you to tell AppleTalk to allow intranode message delivery. This means that two programs running on the same Macintosh can now talk to each other using any of the standard AppleTalk protocols.

Listing 9-4 shows how the **PSelfSend** call works. It's very simple really, but it does require that you use an MPP parameter block and fill it out correctly.

Listing 9-4. Turning SelfSend mode on

```
1: myMPPPB.newSelfFlag := kSelfSendOn;
2: stat := PSetSelfSend(@myMPPPB,kSYNC);
```

Line 1 fills in the **newSelfSend** field of the MPP parameter block with kSelfSendOn. The **newSelfSend** field tells **PSelfSend** what the new state should be: on or off. **kSelfSend**, defined as 1, means sending to yourself should be turned on. A value of zero means it should be turned off.

Line 2 makes the call to **PSelfSend**.

After a call to **PSelfSend**, there is a returned value in the **oldSelfFlag** field of the MPP parameter block which contains the previous setting of the **SelfSend** flag so you can set it back later if you so wish. It is normally okay to leave this turned on.

In the earliest days of **SelfSend**, you could cause a few glitches by turning it on. Now, everybody expects to turn it on and write their programs accordingly. There are even a number of common utility programs that simply turn it on the first chance they get and leave it on thereafter. This means that your code should expect **SelfSend** to be turned on—turning it on yourself shouldn't do any harm.

▶ PGetAppleTalkInfo

Included with the other enhancements added by Phase 2 AppleTalk is a new MPP call, **PGetAppleTalkInfo**, that returns to you a wide variety of AppleTalk information. Some of this information is new with Phase 2 AppleTalk, the rest of it had to be read from low-memory globals, a practice that is now being actively discouraged by Apple.

Like other MPP calls, **PGetAppleTalkInfo** takes two parameters, the address of an MPP parameter block and a Boolean that indicates whether the call is synchronous or asynchronous, and it returns a status value. The **PGetAppleTalkInfo** routine follows.

```
FUNCTION PGetAppleTalkInfo(theMPPPBPtr : MPPPBPtr; async:
boolean) : OSErr;

Using the following fields in the MPP Parameter Block:

  →    ioCompletion      - address of completion routine
  ←    ioResult          - result of operation
  →    version           - always 1 for now
  →    varsPtr           - pointer to MPP variables
  ←    dcePtr            - pointer to MPP device control entry
  ←    portID            - port number
  ←    configuration     - configuration flags
  ←    selfSend          - is selfSend turned on or off
  ←    netLo             - lower bound of network range
  ←    netHi             - upper bound of network range
  ←    ourAddr           - this node's network address
  ←    routerAddr        - router's network address
  ←    numOfPHs          - maximum number of protocol handlers
  ←    numOfSkts         - maximum number of static sockets
  ←    numNBPEs          - maximum number of concurrent NBP
                           requests
  ←    ntQueue           - pointer to names queue
  ←    laLength          - length of data link address
  →    linkAddr          - pointer to data link address buffer
  →    zoneName          - pointer to zone name buffer

Errors Returned:

  noErr (0)              No error.
  paramError (-50)       Unknown version number.
```

ioCompletion contains the address of the completion routine called when the asynchronous version of the trap is used. This should be set to NIL if no completion routine is desired.

ioResult contains the result of the trap when it is finished. During asynchronous operation this field is first set to 1 (denoting that the trap is in process) then set to the final result code when the trap is completed.

version contains the version of **PGetAppleTalkInfo**. This is currently version one.

varsPtr is the pointer to the MPP globals. These variables are private to MPP and can be changed by Apple in the future. You shouldn't depend on these.

dcePtr is the pointer to the device control entry for the .MPP driver.

portID is the port number for the .MPP driver. This is always zero unless the driver is being used by a router.

configuration is a long word containing a variety of flags that describe the state of AppleTalk. Bit 31 is set if the machine is using a node number in the server range. Bit 30 is set if the Apple Internet Router is running on this machine. Bit 15 is set if you are on an extended network. Bit 6 is clear if you have multiple zones available on an extended network.

selfSend is zero if **selfSend** is turned off; otherwise it is one.

netLo is the lower bound of the range of network numbers available if you have an extended network. If you do not have an extended network, this is the network number.

netHi is the upper bound of the range of network numbers available if you have an extended network. If you do not have an extended network, this is the network number.

ourAddr is the 24-bit address for this node. The lower byte contains the node ID and the middle two bytes contain the network number.

routerAddr is the 24-bit address for the last router the machine has talked to. The lower byte contains the node ID of the router and the middle two bytes contain the network number of the router.

numOfPHs is the maximum number of protocol handlers that the .MPP driver allows.

numOfSkts is the maximum number of sockets that the .MPP driver allows for statically assigned sockets.

numOfNBPEs is the maximum number of concurrent NBP operations that this .MPP driver allows.

ntQueue is the pointer to the names table queue.

laLength and **linkAddr** describe the size of location of the buffer for the data link address. The **laLength** returns the actual length of the data link address stored in the buffer.

zoneName is the pointer to a buffer containing the zone name.

noErr is returned when the trap completes normally.

paramError is returned when the version number is invalid.

The following sections show examples of using the **PGetAppleTalkInfo** call to fetch various AppleTalk parameters.

▶ IsSelfSendOn routine using PGetAppleTalkInfo

The **IsSelfSend** routine in Listing 9-5 can be used to determine if a Macintosh can send packets to itself.

Listing 9-5. IsSelfSendOn routine using PGetAppleTalkInfo

```
 1: FUNCTION IsSelfSendOn : BOOLEAN;
 2: VAR
 3:     theMPPPB    : MPPParamBlock;
 4: BEGIN {IsSelfSendOn}
 5:     WITH theMPPPB DO BEGIN
 6:         version     := 1;
 7:         linkAddr    := nil;
 8:         zoneName    := nil;
 9:     END;
10:     IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
11:         THEN BEGIN
12:             IF theMPPPB.selfSend = 0
13:                 THEN BEGIN
14:                     IsSelfSendOn := FALSE;
15:                 END
16:                 ELSE BEGIN
17:                     IsSelfSendOn := TRUE;
18:                 END;
19:         END
20:         ELSE BEGIN
21:             IsSelfSendOn := FALSE;
22:         END;
23: END; {IsSelfSendOn}
```

It begins at lines 6–8 by setting the input parameter block fields to acceptable values. The version is always one and neither the **linkAddress** buffer or the **zoneName** buffers are used, so they are set to NIL.

Line 10 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, line 21 returns FALSE as the result of the function.

As long as the call succeeds, line 12 checks the value of the **selfSend** field in the parameter block. If it is zero, self sending is disabled, so FALSE is returned at line 14; otherwise TRUE is returned at line 21.

▶ AreWeAServer routine using PGetAppleTalkInfo

The **AreWeAServer** routine in Listing 9-6 can be used to determine if a Macintosh has a node ID assigned to it in the server range.

Listing 9-6. AreWeAServer routine using PGetAppleTalkInfo

```
 1: FUNCTION AreWeAServer : BOOLEAN;
 2: CONST
 3:     kSrvAdrBit = 31;
 4: VAR
 5:     theMPPPB    : MPPParamBlock;
 6: BEGIN {AreWeAServer}
 7:     WITH theMPPPB DO BEGIN
 8:         version     := 1;
 9:         linkAddr    := NIL;
10:         zoneName    := NIL;
11:     END;
12:     IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
13:         THEN BEGIN
14:             IF BTst(theMPPPB.configuration,kSrvAdrBit)
15:                 THEN BEGIN
16:                     AreWeAServer := TRUE;
17:                 END
18:                 ELSE BEGIN
19:                     AreWeAServer := FALSE;
20:                 END;
21:         END
22:         ELSE BEGIN
23:             AreWeAServer := FALSE;
24:         END;
25: END; {AreWeAServer}
```

It begins at lines 8–10 by setting the input parameter block fields to acceptable values. The version is always one and neither the **linkAddress** buffer or the **zoneName** buffers are used so they are set to NIL.

Line 12 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, line 23 returns FALSE as the result of the function.

As long as the call succeeds, line 14 checks the server address bit (bit 31) in the configuration flags field of the parameter block. If this bit is set, line 16 returns TRUE; otherwise line 19 returns FALSE.

This code can be adapted to check the router bit, the extended bit, the Apple Internet Router bit, and the one zone bit by simply changing line 14 to check these bits instead of the server bit that is shown in Listing 9-6.

▶ GetNetRange using PGetAppleTalkInfo

The **GetNetRange** routine in Listing 9-7 can be used to determine the range of network numbers available on the local wiring.

Listing 9-7. GetNetRange routine using PGetAppleTalkInfo

```
 1: PROCEDURE GetNetRange(VAR lo,hi : integer);
 2: VAR
 3:     theMPPPB    : MPPParamBlock;
 4: BEGIN {GetNetRange}
 5:     WITH theMPPPB DO BEGIN
 6:         version     := 1;
 7:         linkAddr    := nil;
 8:         zoneName    := nil;
 9:     END;
10:     IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
11:         THEN BEGIN
12:             WITH theMPPPB DO BEGIN
13:                 lo := netLo;
14:                 hi := netHi;
15:             END;
16:         END
17:         ELSE BEGIN
18:             lo := 0;
19:             hi := 0;
20:         END;
21: END; {GetNetRange}
```

It begins at lines 6–8 by setting the input parameter block fields to acceptable values. The version is always one and neither the **linkAddress** buffer or the **zoneName** buffers are used, so they are set to NIL.

Line 10 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, lines 18 and 19 set the variable parameters lo and hi to zero.

As long as the call succeeds, lines 13 and 14 set the var parameters lo and hi to the **netLo** and **netHi** fields in the parameter block.

▶ GetOurAddr using PGetAppleTalkInfo

The **GetOurAddr** routine in Listing 9-8 can be used to get the node ID and network number of your Macintosh.

Listing 9-8. GetOurAddr routine using PGetAppleTalkInfo

```
 1: PROCEDURE GetOurAddr(VAR theNodeID : integer;
 2:                      VAR theNetworkNumber : integer);
 3: VAR
 4:     theMPPPB    : MPPParamBlock;
 5: BEGIN {GetOurAddr}
 6:     WITH theMPPPB DO BEGIN
 7:         version     := 1;
 8:         linkAddr    := NIL;
 9:         zoneName    := NIL;
10:     END;
11:     IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
12:         THEN BEGIN
13:             WITH theMPPPB DO BEGIN
14:                 theNodeID := BAnd(ourAddr,$000000FF);
15:                 theNetworkNumber :=
16:                         BSR(BAnd(ourAddr,$00FFFF00),8);
17:             END;
18:         END
19:         ELSE BEGIN
20:             theNodeID        := 0;
21:             theNetworkNumber := 0;
22:         END;
23: END; {GetOurAddr}
```

It begins at lines 7–9 by setting the input parameter block fields to acceptable values. The version is always one and neither the **linkAddress** buffer or the **zoneName** buffers are used so they are set to NIL.

Line 11 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, lines 20 and 21 set the var parameters **theNodeID** and **theNetworkNumber** to zero.

As long as the call succeeds, lines 14 sets the var parameter theNodeID to the low byte of the **ourAddr** field in the parameter block. This is done by using a logical AND operation (BAnd is a routine provided in MPW Pascal to perform the Bitwise AND operation between two long words) to mask the higher bytes of the long word. This leaves the low byte as the result.

Lines 15 and 16 perform a similar operation of masking the middle two bytes of the long word with the addition of a bitwise shift right by eight bits. This allows the two middle bytes to be shifted into the low bytes and be returned as a regular number.

▶ GetRouterAddr using PGetAppleTalkInfo

The **GetRouterAddr** routine in Listing 9-9 can be used to get the node ID and network number of the last router that your Macintosh has communicated with.

Listing 9-9. GetRouterAddr routine using PGetAppleTalkInfo

```
 1: PROCEDURE GetRouterAddr(VAR theNodeID : integer;
 2:                         VAR theNetworkNumber : integer);
 3: VAR
 4:     theMPPPB    : MPPParamBlock;
 5: BEGIN {GetRouterAddr}
 6:     WITH theMPPPB DO BEGIN
 7:         version     := 1;
 8:         linkAddr    := NIL;
 9:         zoneName    := NIL;
10:     END;
11:     IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
12:         THEN BEGIN
13:             WITH theMPPPB DO BEGIN
14:                 theNodeID := BAnd(routerAddr,$000000FF);
15:                 theNetworkNumber :=
16:                         BSR(BAnd(routerAddr,$00FFFF00),8);
17:             END;
18:         END
19:         ELSE BEGIN
20:             theNodeID        := 0;
21:             theNetworkNumber := 0;
22:         END;
23: END; {GetRouterAddr}
```

It begins at lines 7–9 by setting the input parameter block fields to acceptable values. The version is always one and neither the **linkAddress** buffer or the **zoneName** buffers are used so they are set to NIL.

Line 11 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, lines 20 and 21 set the var parameters, **theNodeID**, and **theNetworkNumber** to zero.

As long as the call succeeds, line 14 sets the variable parameter **theNodeID** to the low byte of the **routerAddr** field in the parameter block. This is done by using a logical AND operation (BAnd is a routine provided in MPW Pascal to perform the Bitwise AND operation between two long words) to mask the higher bytes of the long word. This leaves the low byte as the result.

Lines 15 and 16 perform a similar operation of masking the middle two bytes of the long word with the addition of a bitwise shift right by eight bits. This allows the two middle bytes to be shifted into the low bytes and be returned as a regular number.

▶ GetMaxs using PGetAppleTalkInfo

The **GetMaxs** routine in Listing 9-10 can be used to get the maximum number of protocol handlers, the maximum number of static sockets, and the maximum number of concurrent NBP operations that the current .MPP driver allows.

Listing 9-10. GetMaxs routine using PGetAppleTalkInfo

```
PROCEDURE GetMaxs(VAR theMaxPHs    : integer;
                  VAR theMaxSkts   : integer;
                  VAR theMaxNPBEs  : integer);
VAR
    theMPPPB    : MPPParamBlock;
BEGIN {GetMaxs}
    WITH theMPPPB DO BEGIN
        version     := 1;
        linkAddr    := NIL;
        zoneName    := NIL;
    END;
    IF PGetAppleTalkInfo(@theMPPPB,kSYNC) = noErr
        THEN BEGIN
            WITH theMPPPB DO BEGIN
                theMaxPHs   := numOfPHs;
                theMaxSkts  := numOfSkts;
                theMaxNPBEs := numNBPEs;
            END;
        END
        ELSE BEGIN
            theMaxPHs   := 0;
            theMaxSkts  := 0;
            theMaxNPBEs := 0;
        END;
END; {GetMaxs}
```

It begins at lines 8–10 by setting the input parameter block fields to acceptable values. The version is always one and neither the linkAddress buffer or the zoneName buffers are used so they are set to NIL.

Line 12 makes the call to **PGetAppleTalkInfo** synchronously. If this call fails, lines 21–23 set the var parameters **theMaxPHs**, **theMaxSkts**, and **theMaxNBPEs** to zero.

As long as the call succeeds, lines 15–17 set the var parameters **theMaxPHs**, **theMaxSkts**, and **theMaxNBPEs** to the **maxPHs**, **maxSkts**, and **maxNBPEs** fields in the parameter block.

▶ The Transition Queue

Another new feature in AppleTalk Phase 2 is the transition queue. The transition queue provides you a way to be informed of important AppleTalk events such as the opening or closing of the .MPP driver. This normally indicates that the user is turning on or off the AppleTalk network.

To use the transition queue, you can insert a queue entry. This entry contains a pointer to a routine that will then be called when a transition event occurs. This routine is passed information about the transition event and returns a value to the caller that indicates whether the routine accepts the event or, in some circumstances, rejects it. The routine can also communicate with the rest of your program to perform actions appropriate for the transition event.

You can respond to four predefined transitions:

- open transition
- prepare to close transition
- permission to close transition
- cancel close transition

The open transition is used to indicate that the .MPP driver has just been opened. It isn't called if the .MPP driver is already open and some program opens it again.

The prepare to close transition is called just before the .MPP driver is to be closed. When this transition is received, there is no way to prevent the .MPP driver from closing, but it can be used to perform appropriate actions in preparation for the driver closing.

The permission to close transition is called when a program wants to close the .MPP driver and calls **PATalkClosePrep** to ask permission to do so. Your routine should indicate whether it wants to allow the close request or not. Note: even if you deny permission, the other program may still close the .MPP driver anyway.

The cancel close transition is called after some program has denied permission to close the .MPP driver in response to the prepare to close transition.

Remember that after you receive a permission to close transition, you will always receive either a prepare to close transition (if permission was granted) or a cancel close transition (if permission was denied). This means that you can take certain actions after you receive a prepare to close transition, such as no longer accepting connections or accepting other requests. Then, you can either cancel this action when you get the cancel close transition, or you can continue your shutdown when you get the prepare to close transistion.

▶ Inserting an Entry into the Transition Queue

You insert your own entry into the transition queue using the **LAPAddATQ** call and remove it using **LAPRmvATQ**. These calls are defined as follows:

```
FUNCTION  LAPAddATQ(theATQEntry:  ATQEntryPtr):  OSErr;
FUNCTION  LAPRmvATQ(theATQEntry:  ATQEntryPtr):  OSErr;
```

The **ATQEntryPtr** points to a standard queue element having the following format:

```
ATQEntry  =  RECORD
    qLink     : ATQEntryPtr;
    qType     : INTEGER;
    CallAddr  : ProcPtr;
END;
```

It can also be extended to include additional information such as the following:

```
MyATQEntry  =  RECORD
    qLink      :  ATQEntryPtr;
    qType      :  INTEGER;
    CallAddr   :  ProcPtr;
    myA5Reg    :  longint;
    myDataPtr  :  DataBlockPtr;
END;
```

▶ The Transition Handling Routine

Your routine that handles transitions cannot be written directly in Pascal. For some reason, you must use C-style parameter passing. This means that you either write your routine in C or assembler, or write a small assembler stub that calls a Pascal routine.

The C declaration for your transition handling routine has the following form:

```
OSErr  Transition(long  code,  ATQentryPtr  qElem,  void  *p);
```

The code parameter contains the transition selector with the following values:

0	=	Open Transition
2	=	Prepare to Close Transition
3	=	Permission to Close Transition
4	=	Cancel Close Transition

The **qElement** is a pointer to your queue entry.

The final pointer, **p**, points to different data depending on which transition is happening.

For open transitions it points to the Device Manager parameter block for the open call that is opening MPP. You should not modify data found there, but you can look at it if you want to.

For permission to close transitions, it points to four bytes that you can store a pointer to the name of your application if you are denying permission. This name is displayed in a dialog that informs the user which application denied permission.

For both the prepare to close and the cancel close transitions, it is simply NIL.

Important ▶ You should always return noErr (0) from your transition handling routine unless you are denying a request to close. This is especially important to do even if you run across an unknown transition. This assures that any future additional transitions that may be defined by Apple or other users are handled properly.

Listing 9-11 shows an assembly language transition handling routine that simply calls a Pascal transition handling routine. This code was derived by compiling the small MPW C routine found in Listing 9-12.

Listing 9-11. Assembler glue for a Pascal transition handler

```
ATRANS  PROC           EXPORT

                LINK        A6,#$0000
                MOVEM.L     A3/A4,-(A7)
                MOVEA.L     $0010(A6),A4
                MOVEA.L     $000C(A6),A3
                SUBQ.L      #$2,A7
                MOVE.L      $0008(A6),-(A7)
                MOVE.L      A3,-(A7)
                MOVE.L      A4,-(A7)
                JSR         PTRANSITION
                MOVE.W      (A7)+,D0
                MOVEM.L     -$0008(A6),A3/A4
                UNLK        A6
                RTS

                ENDPROC
```

Listing 9-12. C version of Pascal transition handler glue

```
OSErr TRANSITION(long code, ATQentryPtr qElem, void *p)
{
        return (PTRANSITION(code,qElem,p));
}
```

By the Way ▶

When the transition handler is called, you should observe all the rules of interrupt routines since it is possible that it is being called at interrupt time. This means you should not access unlocked data referenced by a handle or move memory. You also need to restore your own A5 register if you want to access your own global variables.

▶ The Chooser

The Chooser provides a very simple and effective way for your users to select or connect to a network resource. It provides a number of benefits to your users—they are familiar with it from using it to select printers, and all they need to do to install a Chooser interface program is to drop the file into their System folder.

▶ The Chooser User Interface

The Chooser provides a very well-defined and fairly restrictive user interface. You have a single text string across the top that you can set, as well as up to two buttons for the user to press. The Chooser fills in a list of remote NBP entities in the list for you by default, but you can override this behavior if it suits your purposes. This may seem rather basic, but this user interface is often adequate for the basic operations of connecting to a remote program and setting up preferences.

Once the connection is made, you are free to put up your own dialog or series of dialogs to lead the user through any series of operations that you see the need for. The only restriction is that you must use a modal dialog, because there is no mechanism built into the Chooser for handling anything else. The Chooser only calls you once, so you have to grab the opportunity and not let go until your program is finished.

▶ Basic Operation of a Chooser Interface

To add your own icon to the Chooser, you create a file with creator type 'RDEV' and define a number of specific resources that make up an RDEV file (commonly referred to as simply an RDEV). The user places the RDEV file into their System folder and then opens the Chooser Desk Accessory.

Your RDEV is invoked when the user selects your RDEV's icon. The Chooser then loads in your resources, displays the desired buttons and text in the Chooser, and begins a conversation with your code.

This conversation consists of repeated calls to your 'PACK' resource. These calls pass six parameters and return an error result. Listing 9-13 shows the prototype of this call interface. Your code should expect to be called in this manner.

Listing 9-13. Call interface for the 'PACK' code resource

```
FUNCTION YourCode( message  : integer;
                   caller   : integer;
                   objName  : StringPtr;
                   zoneName : StringPtr;
                   p1,p2    : LONGINT) : OSErr;
```

The message parameter indicates which action the Chooser wants you to perform. The other parameters take on various meanings based on the value of the message parameter.

When building a simple Chooser interface, the only message you need to worry about is the button message (18). This message is sent when either of the two buttons are pressed by the user. If the user double-clicks on a list item, it acts as if the left button was pressed and a button message is sent to your code.

When the button message is received, the caller parameter is always set to 1, meaning that the Chooser was the caller. Apple reserves any other values for future expansion. Your RDEV code would normally ignore this parameter.

The **objName** parameter contains the null string when button messages are sent. The **zoneStr** parameter contains the name of the currently selected zone or '*' if there is only one zone present on the network.

The **p1** parameter is filled in with a handle to a List Manager list. This list is the one used by the Chooser to display the selections available to the user. This handle is important not only as a way to get the name of the selected entity in that list, but also because the second column of the list (not visible to the user) is where the Chooser stores the network address for that entity. This must be accessed to get the proper socket address for the selected entity.

Listing 9-14 shows how you can retrieve the address of the selected item using List Manager routines.

Listing 9-14. Retrieving the address block for the selected item

```
theCell := Point(LONGINT(0));

IF LGetSelect(TRUE,theCell,ListHandle(p1))
  THEN {it should always be true};
    theCell.h := 1;
    LGetCell(@ATAddr,SIZEOF(AddrBlock),
                       theCell,ListHandle(p1));
  END;
```

Line 1 sets the variable **theCell** to be [0,0], the top-left cell of the list. This is needed because the following **LGetSelect** call looks for the first selection after the cell you pass in. Using [0,0] instructs **LGetSelect** to find the first and only selection (in this case) in the list.

Line 3 calls **LGetSelect** to find the selected cell. This always succeeds when there is a selected cell.

Line 5 sets **theCell**'s horizontal component to be 1. This actually specifies the second column in the list, with column 0 being the first, and is where the Chooser stores the address of the entity that has the name found in column 0.

Line 6 calls **LGetCell** to retrieve the address data and place it into the **ATAddr** variable.

The **p2** parameter contains two pieces of information, each stored in the low-order or the high-order word of the long word parameter. The low-order word contains either a 1, denoting that the left button was pressed, or a 2, denoting that the right button was pressed. The high-order word contains the modifier bits from the event. These let you check for things like the Shift or Option key being down when the button was pressed.

By the Way ▶ Two words make up one long word. Silly, eh?

When creating a simple Chooser interface that only uses button messages, it can be useful to define a special type that makes accessing the **p2** data easier. You can then declare the **p2** parameter to be of this type. Listing 9-15 shows an example of this.

Listing 9-15. Responding to a button message

```
 1: ModifierBooleans = PACKED RECORD
 2:    filler1          : 0..7;
 3:    controlKeyIsDown  : boolean;
 4:    optionKeyIsDown   : boolean;
 5:    capsLockKeyIsDown : boolean;
 6:    shiftKeyIsDown    : boolean;
 7:    commandKeyIsDown  : boolean;
 8:    filler2           : 0..255;
 9: END;
10:
11: ButtonP2 = RECORD
12:    modifiers    : ModifierBooleans;
13:    whichButton : integer;
14: END;
15:
16: FUNCTION myRDEV (
17:   message,caller    : INTEGER;
18:   objName,zoneName : StringPtr;
19:   p1                : LONGINT;
20:   p2                : ButtonP2) : OSErr;
21: BEGIN
22:   IF message = buttonMsg THEN
23:     IF p2.whichButton = kLeftButtonPressed THEN
24:         IF p2.modifiers.optionKeyIsDown
25:            THEN HandleOptionLeftButton
26:            ELSE HandleLeftButton;
27:     ELSE HandleRightButton;
28: END;
```

Lines 1–9 define the ModifierBooleans record that allows direct access to the modifier key flags in the high-order word of **p2**. By using a packed record, the compiler puts each field in the minimum number of bits.

Lines 11–14 combine the ModifierBooleans record with an integer for checking which button was pressed.

Line 20 shows the substitution of the **ButtonP2** type for the normal longint type with the **p2** parameter. **ButtonP2** must be exactly the same size as longint.

Line 23 shows a check for the left button being pressed.

Line 24 shows a check for the Option key being down.

▶ Configuring a Chooser Interface

Chooser documents contain a number of resources that define how Chooser documents work. Each resource is described below. You can define your own resources for your RDEV in addition to these standard resources. The only restriction is that they must not conflict with the ones defined by the Chooser and your resources must have IDs in the range -4096 through -4065.

The 'STR' Resources

Chooser documents contain the following four required string resources:

- **The 'STR' resource with ID = –4096**—This resource tells the Chooser which NBP type it should use when it looks for network entities on your behalf. For example, if you want it to list all the AppleShare servers in the selected zone, you would have this 'STR' resource contain the string "AFP Server."
- **The 'STR' resource with ID = –4091**—This is the 'STR' resource that defines the message displayed across the top of the Chooser dialog. The message should be simple, as anything long will not fit. Something like "Select an XXX" is often all that is required.

The last two 'STR' resources define the title of the two buttons you have at your disposal:

- **'STR' resource with ID = –4093**—defines the left button title
- **'STR' resource with ID = –4092**—defines the right button title

Again, space is at a premium so these must be short button titles.

Note ▶ **There is another STR resource that the Chooser uses—STR with resource ID = –4090. It is used internally by the Chooser and should not be used by your RDEV.**

Listing 9-16 shows Rez source codes that define each of these 'STR' resources.

Listing 9-16. Rez definition for 'STR' resources

```
resource 'STR ' (-4096, purgeable) {
  "Example RDEV"
};
resource 'STR ' (-4093, purgeable) {
  "Connect"
};
resource 'STR ' (-4092, purgeable) {
  "Customize'
};
resource 'STR ' (-4091, purgeable) {
  "Select a Target:"
};
```

The 'GNRL' Resource

There is a single 'GNRL' resource defined by the Chooser. This is the 'GNRL' resource with ID = –4096 and it contains 2 bytes that define the timeout information for the NBP lookup performed by the Chooser. The first byte is the interval and the second byte is the retry count. These two values correspond to the equivalent values in the NBP PLookup call.

Listing 9-17 shows Rez source code that defines a 'GNRL' resource that instructs the Chooser to use a timeout value of 5 and 3 retries.

Listing 9-17. Rez definition for the 'GNRL' resource

```
data 'GNRL' (-4096, purgeable) {
 $"05 03"
};
```

The 'nrct' Resource

There is a single 'nrct' resource defined by the Chooser. It defines the bounding rectangles for the left and right buttons. The 'nrct' resource with ID = –4096 contains a two-item list of rectangle coordinates—the first for the left button, and the second for the right button.

Listing 9-18 shows Rez source code that defines an 'nrct' resource that places the left and right buttons.

Listing 9-18. Rez definition for the 'nrct' resource

```
resource 'nrct' (-4096) {
 {
  { 112, 206, 132, 266 },
  { 112, 286, 132, 364 }
}
```

The 'LDEF' Resource

The Chooser uses the standard List Manager 'LDEF' to display the names it finds when it does the NBP lookup. It is possible to substitute your own 'LDEF' for the standard one by including it as 'LDEF' with resource ID = –4096.

Substituting your own 'LDEF' lets you present a more informative or aesthetic list than the default one. For example, your 'LDEF' could display a small icon next to each name to indicate something about the remote entities involved or display the names in different colors or typefaces.

The 'PACK' Resource

The key resource in an RDEV is the 'PACK' resource with ID = –4096. This resource contains the code that implements the functionality of your RDEV. Because it is a 'PACK' code resource, it must conform to the standard structure of a 'PACK' resource.

'PACK' resources have a header consisting of 16 bytes of data that are interpreted by the Chooser to define the RDEV's characteristics. Figure 9-1 shows a 'PACK' header used by an RDEV.

$60	$0E	branch over header (BRA.S 16)
$00	$80	device id
$50	$41	'PACK' constant
$43	$4B	
$F0	$00	-4096 constant
$00	$01	version number
$8E	$00	flags (longword)
$00	$00	

Figure 9-1. 'PACK' header structure

Listing 9-19 shows the 68000 assembly language source code that produced the header. It begins with a branch instruction on line 1. This branch instruction will skip over the data portions of the header. This is done because the Chooser jumps to the beginning of the 'PACK' code resource to invoke it.

Listing 9-19. Source code for a 'PACK' header

```
1: BRA.S        @1
2: DC.W         80
3: DC.L         ('PACK')
4: DC.W         $F000
5: DC.W         1
6: DC.L         %10001110000000000000000000000000
```

Line 2 is a word containing the device ID. This provides an identifier for your RDEV to the Chooser.

Lines 3 and 4 define two constants consisting of the four characters 'P', 'A', 'C', 'K', and the number –4096.

Line 5 defines a version number for the 'PACK' resource.

Line 6 defines a long word of flag bits. The values for these flags are detailed in Figure 9-2.

The flags found in the 'PACK' resource are used by the Chooser to determine the specific behavior of the RDEV. These flags control such things as which messages are received by the RDEV, which buttons are actually used by the RDEV, if multiple selections are supported, and how zone names are handled.

Simple RDEVs only set two to four of the flags, but more sophisticated RDEVs will make use of some of the other flags. The four flags set by simple RDEVs are the following:

- Bit 25 Simple RDEVs don't save zone names
- Bit 26 Simple RDEVs sometimes use the right button
- Bit 27 Simple RDEVs sometimes use the left button
- Bit 31 Simple RDEVs do use AppleTalk

Figure 9-2 lists all of these flags with a short description of their use. Further information can be found in *Inside Macintosh*, Volume IV.

Bit #	Explanation
0	Reserved.
1	Reserved.
2	Reserved.
3	Reserved.
4	Reserved.
5	Reserved.
6	Reserved.
7	Reserved.
8	Reserved.
9	Reserved.
10	Reserved.
11	Accepts terminate messages.
12	Accepts deselect messages.
13	Accepts select messages.
14	Accepts getSel messages.
15	Accepts fillList messages.
16	Accepts newSel messages.
17	Reserved.
18	Reserved.
19	Reserved.
20	Reserved.
21	Reserved.
22	Reserved.
23	Reserved.
24	Uses actual zone names.
25	No zone name is saved.
26	Uses right button.
27	Uses left button.
28	Multiple selections allowed.
29	Reserved.
30	Reserved.
31	Uses AppleTalk.

Figure 9-2. RDEV flags

By the Way ▶

A common problem when debugging RDEV code is that the Chooser doesn't seem to notice changes made to an RDEV as you change it in your development process. This happens because the Chooser caches the RDEV's flag information and does not look at the RDEV file each time it is invoked, even if you have just updated the file. To get around this troublesome behavior, you must open the Chooser with both the Command and Option keys down. This instructs the Chooser to discard its cache of RDEV characteristics and examine each RDEV file again. If you do this successfully, the Chooser will beep at you.

▶ Summary

This chapter described a variety of AppleTalk topics not covered in other parts of this book. First the issue of opening the various AppleTalk drivers was covered—how and why you should open the .MPP driver, the .XPP driver, and the .DSP driver were discussed and illustrated. Then the **PSelfSend** and **PGetAppleTalkInfo** calls were described. You were shown how to allow your program to talk to other programs on the same Macintosh using the **PSelfSend** call and how to get information about AppleTalk using the **PGetAppleTalkInfo** call. The transition queue was also described. You were shown how to have AppleTalk inform you about important AppleTalk events.

Finally, the Chooser was discussed. Its defining resources were examined and examples were shown of each of these resources. The flags found in the 'PACK' code resource were also discussed.

The final section of this book follows. It offers detailed walkthroughs of AppleTalk programs. Chapter 10 is about NameTool—the Macintosh Programmer's Workshop (MPW) NBP tool.

PART THREE

▶ AppleTalk Programming Examples

Part Three of this book walks you through three real AppleTalk programs. These are meant to be basic models that demonstrate how to put the information in the previous chapters to work in a working program. You should be able to take these examples and build your own programs using many of the same techniques.

- Chapter 10 discusses the NameTool. The NameTool illustrates how to use NBP, ZIP, and ATP to get information about names used on the network. It does this in the context of an MPW tool.
- Chapter 11 discusses the RemoteSysInfo RDEV. It uses NBP and ATP to exchange information over the network using a simple client server model. It describes how the three pieces of RemoteSystemInfo—the INIT, the resident code, and the RDEV Chooser interface—are built.
- Chapter 12 discusses the network checkers game. It uses NBP and ADSP in a MacApp framework to exchange game data between two players running the game on different Macintoshes on an AppleTalk network. It explains the object classes used to work with asynchronous ADSP operations.

10 ► NameTool: An AppleTalk MPW Tool for Using NBP Lookups

NameTool is a Macintosh Programmer's Workshop (MPW) tool that allows you to explore the use of the Name Binding Protocol and the Zone Information Protocol. It is invoked by entering it on the MPW command line. The results are directed to the standard MPW output.

NameTool has a very simple console-oriented interface that leads to a very simple program structure that doesn't interfere with understanding the AppleTalk code.

► Goals for NameTool

NameTool illustrates a number of basic AppleTalk programming techniques. It uses ZIP to look up the zones on the current network and NBP to look up names in each zone. It handles both Phase 1 and Phase 2 AppleTalk networks, illustrating how to use ZIP in each of these environments. If you expect your programs to operate in only one of these environments, or the other, you can simplify your own programs by just using one approach.

NameTool is implemented as an MPW tool, but the code can be easily adapted to other environments, such as Symantec's THINK Pascal.

▶ How to Use NameTool

The basic operation performed by **NameTool** is looking up NBP names on your AppleTalk network. By default, **NameTool** looks up all names in all the zones found in your network. A number of options are provided to give more control over its operation.

NameTool deals with the following three types of options:

- The help option
- Specifying which names to look up
- Controlling the NBP and ATP timeout and retry behavior

▶ The Help Option

The help option is invoked by typing "–h" on the command line. When this option is used, **NameTool** prints out a small help message that tells the user how to use the other options. Listing 10-1 shows **NameTool** used with the help option and its output.

Listing 10-1. Output from the help option

```
NameTool -h
# NameTool 1.0 - Looks at network names.
#   -z  {zoneName}     Only looks in specified zone.
#   -n  {serverName}   Only looks for servers with specified
                       name.
#   -t  {typeName}     Only looks for types with specified
                       name.
#   -i  {interval}     Interval for NBPLookup call.
#   -c  {count}        Count for NBPLookup call.
#   -m                 Use local zone.
#   -?                 Prints this text.
```

▶ The Zone, Name, and Type Options

NameTool provides the following three options to specify which names to look up on the network:

- **The zone option**—invoked by "–z" followed by the zone name, is used to indicate specific zones that you want to look up
- **The name option**—invoked by "–n" followed by the name, is used to indicate specific names that you want to look up

- **The type option**—invoked by "–t" followed by the type, is used to indicate specific types that you want to look up

These options can be combined to look up, say, a specific name in a specific zone, or certain types with a certain name. These options can also contain appropriate wildcard characters to further refine what you are looking up.

The zone, name, and type options correspond directly to the three parts of the NBP name used by the **PLookupName** trap. This means that they must conform to the rules that govern NBP names. That is, they must be less than or equal to 31 characters in length. They can also contain the wildcard characters that **PLookupName** handles. See Chapter 5 for details.

There is a related option, "–m", that overrides the zone option. It specifies that only the local zone should be looked at ("–m" stands for My zone).

▶ The Interval and Timeout Options

Two options are provided for controlling the timing used by the NBP **PLookup** call, the ZIP calls to the .XPP driver, and the ATP calls used to access ZIP in Phase 1 AppleTalk networks. These two options are: the timeout option (invoked by "–t" followed by the timeout value) and the interval option (invoked by "–c" followed by the interval count). The interval option is used to specify the interval count used in these calls. The timeout option is used to specify the timeout used in these calls. These options correspond directly to the **timeout** and **retry** fields in the various parameter blocks used.

▶ General Comments About NameTool

NameTool makes use of the synchronous variations of the AppleTalk calls. It could have used the asynchronous variations, and instead of locking up the Macintosh while waiting for the completion of the calls, it could have made the calls asynchronously and yielded time to other processes running. The synchronous variation was used here only to simplify the example.

Another important consideration in **NameTool** is that it uses AppleTalk Phase 2 features if it can; otherwise it uses techniques appropriate for Phase 1. This ability to use AppleTalk Phase 2, if available, comes up in two places: when getting the current zone name and when getting a list of all known zones. In both of these cases the code checks for an AppleTalk version number and uses the Phase 2 approach if it is greater than or equal to version 53.

▶ Sample of the NameTool Output

Listings 10-2, 10-3, and 10-4 show three examples of using **NameTool** to look at a network. Each command was issued on the same AppleTalk network. The network is small—only two zones and four machines—but sufficient for the purpose of seeing how **NameTool** works.

Listing 10-2 shows the output from issuing the **NameTool** command without any options specified. It shows every NBP name found in the entire network. There were three Macintoshes and one LaserWriter running a variety of network software.

The format of the output is NBP name, followed by the type, zone, network, node, and socket numbers.

Listing 10-2. NameTool results #1

```
NameTool
"Router SE" "Public Foldert" "Downstairs" net=2 node=158 socket=251
"Router SE" "Example RDEV" "Downstairs" net=2 node=158 socket=252
"Router SE" "AppleRouter" "Downstairs" net=2 node=158 socket=253
"Michael's Mac IIci" "Macintosh" "Upstairs" net=1 node=9 socket=251
"Michael's Mac IIci" "Public Foldert" "Upstairs" net=1 node=9
socket=252
"Michael's Mac IIci" "Example RDEV" "Upstairs" net=1 node=9 socket=253
"Kathleen's Mac IIcx" "Macintosh IIcx" "Upstairs" net=1 node=122
socket=252
"Kathleen's Mac IIcx" "Example RDEV" "Upstairs" net=1 node=122
socket=253
"South Bay NTX " "LaserWriter" "Upstairs" net=1 node=190 socket=250
___________
 2 zones
 9 NBP names
```

Listing 10-3 shows the output from issuing the **NameTool** command using the name option. It specifies that only NBP names with the name "Router SE" should be shown. The results illustrate three names that match.

Listing 10-3. NameTool results #2

```
NameTool -name 'Router SE'
"Router SE" "Public Foldert" "Downstairs" net=2 node=158 socket=251
"Router SE" "Example RDEV" "Downstairs" net=2 node=158 socket=252
"Router SE" "AppleRouter" "Downstairs" net=2 node=158 socket=253
____________
 2 zones
 3 NBP names
```

Listing 10-4 shows the output from issuing the **NameTool** command using the type option. It specified that only NBP names with the type "Example RDEV" should be shown. The results show three matching names.

Listing 10-4. NameTool results #3

```
NameTool -type 'Example RDEV'
"Router SE" "Example RDEV" "Downstairs" net=2 node=158
socket=252
"Michael's Mac IIci" "Example RDEV" "Upstairs" net=1 node=9
socket=253
"Kathleen's Mac IIcx" "Example RDEV" "Upstairs" net=1 node=122
socket=253
____________
 2 zones
 3 NBP names
```

▶ The Structure of NameTool

NameTool has a very simple structure. It consists of a single source file containing the main program with a small number of supporting subroutines. These supporting subroutines fall into two major categories: general support routines and network access support routines.

The general support routines contained in **NameTool** perform general housekeeping functions such as parsing the MPW command line, checking for error return values, and handling strings. These routines can be used in other MPW tools and do not rely on AppleTalk for their operation. See "The General Routines" section later in this chapter for details.

The network access support routines do the work of accessing AppleTalk. They use NBP, ATP, and ZIP to accomplish their tasks and are very self-contained. They can be used in other programs with little modification. Do keep in mind that they are oriented toward a synchronous approach and that most programs usually call for using an asynchronous approach. See "The AppleTalk Routines" section later in this chapter for details.

Figure 10-1 lists the routines used by **NameTool** and what category of routine they fall into.

Routine Name	Routine Category
CheckStat	General Routine
GetParm	General Routine
StrFromPtr	General Routine
PrintHelp	General Routine
GetMyZone	AppleTalk Routine
GetMyZone1	AppleTalk Routine
GetMyZone2	AppleTalk Routine
BuildZoneList	AppleTalk Routine
BuildZoneList1	AppleTalk Routine
BuildZoneList2	AppleTalk Routine
ProcessZone	AppleTalk Routine

Figure 10-1. Routines found in NameTool

▶ The Header Section of NameTool

Listing 10-5 shows the header section of **NameTool**. It contains the program statement, constants, types, and variables used by the entire program.

Listing 10-5. Header section of NameTool

```
PROGRAM NameTool;

USES
     OSIntf, AppleTalk, PasLibIntf, Packages,
     IntEnv, CursorCtl;

CONST

```

```
 9:     kSYNC    = FALSE;
10:     kASYNC   = TRUE;
11:
12: TYPE
13:     xCallParam = PACKED RECORD
14:         qLink:QElemPtr;
15:         qType:INTEGER;
16:         ioTrap:INTEGER;
17:         ioCmdAddr:Ptr;
18:         ioCompletion:ProcPtr;
19:         ioResult:OSErr;
20:         ioNamePtr:StringPtr;
21:         ioVRefNum:INTEGER;
22:         ioRefNum:INTEGER;
23:         csCode:INTEGER;
24:         xppSubCode:INTEGER;
25:         xppTimeOut:Byte;
26:         xppRetry:Byte;
27:         filler:INTEGER;
28:         zipBuffPtr:Ptr;
29:         zipNumZones:INTEGER;
30:         zipLastFlag:INTEGER;
31:         zipInfoField:  PACKED ARRAY [1..70] of Byte;
32:     END; {xCallParam}
33:
34: VAR
35:     nameCount,
36:     zoneCount         : integer;
37:
38:     tempStr,
39:     zoneParm,
40:     typeParm,
41:     nameParm          : str255;
42:
43:     intervalNumber    : longint;
44:     countNumber       : longint;
```

Listing 10-5 begins in the traditional Macintosh manner with a program statement at line 1 and a USES section at lines 3–5 that reference the units that the tool makes use of.

Two global constants are defined at lines 7–11. kSYNC and kASYNC are used rather than TRUE and FALSE to aid in readability.

Lines 12–32 define the type xCallParam. This is a parameter block used by the ZIP calls to the .XPP driver. xCallParam needs to be defined here because it is not included in the MPW 3.1 interface files. Later versions of MPW will probably provide this record definition; when that happens, you should use the definition provided and remove the old definition from the code.

Lines 34–44 define the global variables used by **NameTool**. One of these variables, **tempStr**, is simply that; a temporary string used when parsing the command line. It must be global since it is used in the main routine.

The **nameCount** and **zoneCount** variables are used to accumulate the total number of zones and names found during the lookup process. These statistics are reported to the user when the program is finished.

The rest of the global variables are filled in with values from the command line to control the operation of the tool. If no value is specified for them on the command line, a default value is used.

▶ The General Routines

The general routines used by NameTool are not specific to **NameTool**; they are general purpose routines that can be used in any MPW tool or program.

CheckStat

CheckStat is a very simple procedure that checks the value passed in against **noErr** (0). If any other value besides **noErr** is encountered, **CheckStat** beeps to get your attention and writes the offending error value and a user-supplied message to the standard output.

CheckStat can be modified in a number of ways to provide additional value to you. For example, adding a call to the Debugger trap will drop you into Macsbug (or your debugger of choice) to let you inspect the environment when the error occurred. Listing 10-6 shows the **CheckStat** routine.

Listing 10-6. The CheckStat routine

```
PROCEDURE CheckStat(status: OSErr; displayText: str255);
BEGIN {CheckStat}
    IF status <> noErr
        THEN BEGIN
            SysBeep(5);
            writeln('CheckStat: stat = ',
                        status:1,', ',DIsplayText);
        END;
END; {CheckStat}
```

GetParm

GetParm is a function that checks for the existence of a specific option on the command line. The options consist of a dash followed by a single character, then another string. An example of this is the zone option in **NameTool**. It has the form "–z zonename". You specify which character you are looking for in the first parameter, and if there is such an option on the command line, the function returns TRUE and passes the other string back in the second parameter.

The algorithm used assumes there is no match, then checks each argument of the command line to see if it has a "–" followed by the specified single character. If it finds a match, it returns the next argument if one exists. Listing 10-7 shows the **GetParm** routine.

Listing 10-7. The GetParm routine

```
FUNCTION GetParm(c : char; var s : str255) : boolean;
VAR
    i : integer;
BEGIN {GetParm}
    s := '';
    GetParm := FALSE;
    i := 1;
    WHILE i <= argC DO BEGIN
        IF (argV^[i]^[1] = '-') and (argV^[i]^[2] = c)
            THEN BEGIN
                IF i < argC
                    THEN BEGIN
                        s := ArgV^[i+1]^;
                    END;
                GetParm := TRUE;
                Exit(GetParm);
            END
            ELSE BEGIN
                i := i + 1;
            END;
    END;
END; {GetParm}
```

StrFromPtr

StrFromPtr is a function that returns the string pointed to by the pointer, p. It is useful for copying strings out of buffers, such as the zone names from the zone name buffer passed back from the ZIP calls. Listing 10-8 shows the **StrFromPtr** routine. The length of the string is also specified.

Listing 10-8. The StrFromPtr routine

```
 1: FUNCTION StrFromPtr(p : Ptr; len : integer) : str255;
 2: TYPE
 3:     CharPtr = ^char;
 4: VAR
 5:     tStr    : str255;
 6:     i       : integer;
 7: BEGIN {StrFromPtr}
 8:
 9:     tStr[0] := chr(len);
10:     for i := 1 to len do BEGIN
11:         tStr[i] := CharPtr(p)^;
12:         p := Ptr(ORD4(p)+1);
13:     END;
14:
15:     StrFromPtr := tStr;
16:
17: END; {StrFromPtr}
```

The algorithm in line 9 starts by putting the length into the first byte of the string that is being built up. This is because Pascal strings store their length in this location.

The rest of the string is filled in using the loop in lines 10-13. The character is copied from the memory pointer pointed to by the pointer into the string. As the loop repeats, the pointer is moved to the next byte of memory.

Finally in line 15, the temporary string is assigned to the function so it will be returned as the value of the function.

PrintHelp

PrintHelp is a very simple procedure that writes the help text to the standard MPW output. Change this text if you change the tool to operate differently. Listing 10-9 shows the **PrintHelp** routine.

Listing 10-9. The PrintHelp routine

```
PROCEDURE PrintHelp;
BEGIN {PrintHelp}
    write  ('# NameTool 1.0 - Looks ');
    writeln('at network names.');
    write  ('#   -z  {zoneName}     ');
    writeln('Only looks in specified zone.');
    write  ('#   -n  {serverName}   Only looks ');
    writeln('for servers with specified name.');
    write  ('#   -t  {typeName}     Only looks ');
    writeln('for types with specified name.');
    write  ('#   -i  {interval}     ');
    writeln('Interval for NBPLookup call.');
    write  ('#   -c  {count}        ');
    writeln('Count for NBPLookup call.');
    write  ('#   -m                 ');
    writeln('Use local zone.');
    write  ('#   -?                 ');
    writeln('Prints this text.');
END; {PrintHelp}
```

▶ The AppleTalk Routines

The AppleTalk routines that **NameTool** uses perform some very common AppleTalk operations. They can be adapted for use on your own programs with little change. The only real shortcoming of these routines is that they call AppleTalk synchronously and your programs should probably call them asynchronously.

GetZIPAddr

GetZIPAddr is a function that passes back the address of your local router. The function returns FALSE if no local router is available, or TRUE if there is one. Listing 10-10 shows the **GetZIPAddr** routine.

Listing 10-10. The GetZIPAddr routine

```
 1: FUNCTION GetZIPAddr(VAR ZIPAddr : AddrBlock) : BOOLEAN;
 2: CONST
 3:     kZIPSocket = 6;
 4: VAR
 5:     theNode       : integer;
 6:     theNet        : integer;
 7:     theRouterNode : integer;
 8: BEGIN {GetZIPAddr}
 9:     GetZIPAddr := FALSE;
10:     IF GetNodeAddress(theNode, theNet) <> noErr
11:         THEN BEGIN
12:             EXIT(GetZIPAddr);
13:         END;
14:     theRouterNode := GetBridgeAddress;
15:     IF (theRouterNode = 0)
16:         THEN BEGIN {No zones}
17:             EXIT(GetZIPAddr);
18:         END;
19:     WITH ZIPAddr DO BEGIN
20:         aNet    := theNet;
21:         aNode   := theRouterNode;
22:         aSocket := kZIPSocket;
23:     END;
24:     GetZIPAddr := TRUE;
25: END; {GetZIPAddr}
```

The first step to finding the local router is to get the network number. This is retrieved easily by using the **GetNodeAddress** call as shown on line 10. **GetNodeAddress** normally shouldn't fail, but if it does, it usually means that the .MPP driver hasn't been opened yet.

Next, you want to get the node address of the router using the **GetBridgeAddress** routine shown on line 14. If this routine returns 0, you know there are no routers out there.

Because you know that the ZIP socket is always 6, you don't have to do anything more—you have the address. Lines 19–23 collect the three components of the router's address into the **ZIPAddr** that is returned to the call.

If you get to line 24, you know that you have assembled the router's address; returning TRUE will signal success to the calling routine.

GetMyZone

GetMyZone is a simple dispatching routine. It checks to see which version of AppleTalk is running. If it finds AppleTalk version 53 or later it calls the Phase 2 implementation **GetMyZone2**. If a Phase 1 driver is present, **GetMyZone1** is called. Listing 10-11 shows the **GetMyZone** routine.

Listing 10-11. The GetMyZone routine

```
FUNCTION GetMyZone : str255;
VAR
    thisMac : SysEnvRec;
BEGIN {GetMyZone}
    IF SysEnvirons(1, thisMac) = noErr
        THEN BEGIN
            IF thisMac.atDrvrVersNum >= 53
                THEN BEGIN
                    GetMyZone := GetMyZone2;
                END
                ELSE BEGIN
                    GetMyZone := GetMyZone1;
                END;
        END
        ELSE BEGIN
            GetMyZone := '*';
        END;
END; {GetMyZone}
```

GetMyZone1

GetMyZone1 is the Phase 1 implementation of the **GetMyZone** function. To get the local zone name, it constructs a ZIP request and sends it (using ATP) to the local router. Then, it returns the zone name that comes back in the response message. Listing 10-12 shows the **GetMyZone1** routine.

Listing 10-12. The GetMyZone1 routine

```
FUNCTION GetMyZone1 : str255;
CONST
    kMyZoneCall = $07000000;
VAR
    stat             : OSErr;
    theATPPBptr      : ATPPBptr;
    theBDS           : BDSElement;
    resultBuff       : str255;
    currZonePtr      : Ptr;
    index, count     : integer;
    ignore           : integer;
    theATPPB         : ATPParamBlock;
    theZIPAddr       : AddrBlock;
BEGIN {GetMyZone1}
    GetMyZone1 := '*';
    IF NOT GetZIPAddr(theZIPAddr)
        THEN BEGIN
            EXIT(GetMyZone1);
        END;
    WITH theBDS DO BEGIN
        buffSize := 255;
        buffPtr := @resultBuff;
    END;
    WITH theATPPB DO BEGIN
        atpFlags   := 0;
        addrBlock  := theZIPAddr;
        reqLength  := 0;
        reqPointer := NIL;
        bdsPointer := @theBDS;
        numOfBuffs := 1;
        timeOutVal := intervalNumber;
        retryCount := countNumber;
        userData   := kMyZoneCall;
    END;
    IF PSendRequest(@theATPPB, kSYNC) = noErr
        THEN BEGIN
            GetMyZone1 :=
                StrFromPtr(@resultBuff,Ptr(@resultBuff)^);
        END;
END; {GetMyZone1}
```

It begins by assuming that there are no zone names found and sets the function result to be '*'. If something happens to indicate that no zone name exists, the function need only exit and the return values will already be correct for that case.

Lines 16–19 attempt to get the router's address. If this fails, it goes with the previous assumption and exits.

Next it builds a BDS on lines 20–23. It uses a convenient variable of type str255 as the result buffer because the result is guaranteed to be smaller than this.

Then lines 24–34 fill in the ATP parameter block in preparation for sending the ATP request to the router. Note the **userBytes** being set to the constant $07000000. The high-byte contains 7 bytes, indicating that the message is requesting the local zone's name.

The ATP request is sent on line 35 using a synchronous call to **PSendRequest**. If no error is encountered, lines 37 and 38 set the return value to the zone name found in the ATP response message.

GetMyZone2

GetMyZone2 is the Phase 2 implementation of the **GetMyZone** function. To get the local zone name, it makes a call to the .XPP driver with the subcode set to **zipGetMyZone**.

It begins by assuming that there are no zone names found and sets the function result to be '*' denoting this. If something comes up indicating that no zone name exists, the function need only exit and the return values will already be correct for that case. Listing 10-13 shows the **GetMyZone2** routine.

Listing 10-13. The GetMyZone2 routine

```
 1: FUNCTION GetMyZone2 : str255;
 2: CONST
 3:     xCall           = 246;
 4:     zipGetMyZone    = 7;
 5: VAR
 6:     theXPBPB        : xCallParam;
 7:     xppDriverRefNum : integer;
 8:     returnedZoneName: str255;
 9: BEGIN {GetMyZone2}
10:     GetMyZone2 := '*';
11:     IF OpenDriver('.XPP', xppDriverRefNum) <> noErr
12:         THEN EXIT(GetMyZone2);
13:     WITH theXPBPB DO BEGIN
14:         zipInfoField[1] := 0;
15:         zipInfoField[2] := 0;
16:         zipLastFlag := 0;
17:         ioRefNum    := xppDriverRefNum;
18:         csCode      := xCall;
19:         xppSubCode  := zipGetMyZone;
20:         xppTimeOut  := intervalNumber;
21:         xppRetry    := countNumber;
22:         zipBuffPtr  := @returnedZoneName;
23:     END;
24:     IF PBControl(@theXPBPB,kSYNC) = noErr
25:         THEN BEGIN
26:             GetMyZone2 := returnedZoneName;
27:         END;
28: END; {GetMyZone2}
```

The .XPP driver is opened on line 11. If any error is encountered, it exits.

Next, on lines 13–23, it fills in the XPP parameter block with the values needed. Note on line 19 that the **subCode** field is being set to **zipGetMyZone** to tell the .XPP driver which action should really be called.

If the **PBControl** call returns no error on line 24, line 26 sets the function's return to be the zone name the .XPP driver returned in **returnedZoneName**.

BuildZoneList

BuildZoneList is a simple dispatching routine. It checks to see which version of AppleTalk is running; if it finds AppleTalk version 53 or later, it calls the Phase 2 implementation **BuildZoneList2**. If a Phase 1 driver is present, **BuildZoneList1** is called. Listing 10-14 shows the **BuildZoneList** routine.

Listing 10-14. The BuildZoneList routine

```
PROCEDURE BuildZoneList;
VAR
    thisMac : SysEnvRec;
BEGIN {BuildZoneList}
    IF SysEnvirons(1, thisMac) = noErr
        THEN BEGIN
            IF thisMac.atDrvrVersNum >= 53
                THEN BEGIN
                    BuildZoneList1;
                END
                ELSE BEGIN
                    BuildZoneList2;
                END;
        END;
END; {BuildZoneList}
```

BuildZoneList1

BuildZoneList1 is the Phase 1 implementation of the **BuildZoneList** function. To get the zone list it constructs a ZIP request and sends it (using ATP) to the local router. Then it extracts the zone name from the response packet and passes the zone name to the **ProcessZone** routine. It repeats this process until all zones have been fetched and processed. Listing 10-15 shows the **BuildZoneList1** routine.

Listing 10-15. The BuildZoneList1 routine

```
PROCEDURE BuildZoneList1;
CONST
    kGZLCall    = $08000000;
    kZonesSize  = 578;
TYPE
    GetZoneUserBytes = PACKED RECORD
        IsLastPacket: BOOLEAN;
        Filler      : 0..255;
        ZoneCount   : integer;
    END;
VAR
    theBDS          : BDSElement;
    zonePtr         : Ptr;
    currZonePtr     : Ptr;
    index, count    : INTEGER;
    zoneName        : str255;
    theATPPB        : ATPParamBlock;
    theZIPAddr      : AddrBlock;
BEGIN {BuildZoneList1}
    IF zoneParm <> '*'
        THEN BEGIN
            ProcessZone(zoneParm);
            zoneCount := 1;
            EXIT(BuildZoneList1);
        END;
    IF NOT GetZIPAddr(theZIPAddr)
        THEN BEGIN
            ProcessZone('*');
            zoneCount := 1;
            EXIT(BuildZoneList1);
        END;
    zonePtr := NewPtr(kZonesSize);
    IF zonePtr = NIL
        THEN EXIT(BuildZoneList1);
    WITH theBDS DO BEGIN
        buffSize := kZonesSize;
        buffPtr := zonePtr;
    END;
    WITH theATPPB DO BEGIN
        numOfBuffs := 1;
        atpFlags   := 0;
        reqLength  := 0;
```

```
43:         reqPointer := NIL;
44:         bdsPointer := @theBDS;
45:         addrBlock  := theZIPAddr;
46:         timeOutVal := intervalNumber;
47:         retryCount := countNumber;
48:     END;
49:     index := 1;
50:     count := 0;
51:     SpinCursor(32);
52:     REPEAT
53:         theATPPB.userData := kGZLCall + index;
54:         CheckStat(PSendRequest(@theATPPB,
55:             kSYNC), 'PSendRequest failed');
56:         count := count +
57:             GetZoneUserBytes(theBDS.userBytes).ZoneCount;
58:         currZonePtr := zonePtr;
59:         REPEAT
60:             zoneName :=
61:                     StrFromPtr(currZonePtr,currZonePtr^);
62:             IF (zoneName = zoneParm) OR (zoneParm = '*')
63:                 THEN BEGIN
64:                     ProcessZone(zoneName);
65:                 END;
66:             zoneCount := zoneCount + 1;
67:             currZonePtr := Ptr(LONGINT(currZonePtr) +
68:                                 currZonePtr^+1);
69:             index := index + 1;
70:         UNTIL index > count;
71:     UNTIL GetZoneUserBytes(theBDS.userBytes).IsLastPacket;
72:     IF zonePtr <> NIL THEN
73:         DisposPtr(zonePtr);
74: END; {BuildZoneList1}
```

Listing 10-15 begins by checking to see if a specific zone has been indicated. Line 20 checks this, and if only one zone has been specified, it is processed alone and the routine exits.

Line 26 attempts to get the address of the router. If none is found, only the local zone is processed by passing **ProcessZone** the string '*'.

Lines 32–34 attempt to allocate a buffer for response data from the router. It's large enough to contain an entire response packet. If the call to **NewPtr** fails, the routine bails out by simply exiting.

Lines 35–38 set up the BDS for the response message using the buffer allocated in lines 32–34.

Lines 39–48 fill in the ATP parameter block with the required values. Note that the **atpFlags** are set to zero, requesting that XO (exactly once) mode not be used. Also note that no request message data is sent; just the **userData**.

Lines 49 and 50 set up the loop.

Line 51 calls **SpinCursor** to yield some time to other processes and spin the MPW cursor, providing feedback to the user that work is being done.

Line 53 sets the proper **userData** to indicate to the router that a zone list request is being made and that the index into the zone names should be the value of the index variable. This starts out as 1, then increases each time through the loop.

Line 54 makes the call to **PSendRequest** to send the ATP request to the router.

Line 56 increments the zone count by the number of zones that came in the current response packet, and line 57 sets the **currZonePtr** to point to the beginning of the zone names buffer.

The loop that goes from lines 56–70 processes the zones in the response packet.

Line 60 copies the zone name found at the current pointer position into **zoneName**, and this is fed into the **ProcessZone** routine at line 64, as long as it either matches the specific zone requested, or no specific zone was specified.

Lines 66–68 do the loop maintenance. The **currZonePtr** is bumped to point to the next zone name and the zone index is incremented to keep track of how many zone names have been processed.

Line 70 allows the inner loop to exit when all the zone names in the response packet have been processed.

Line 71 allows the outer loop to exit when the response packet has the **IsLastPacket** flag set to TRUE.

Line 72 cleans up by deallocating the memory used to store the response packet.

BuildZoneList2

BuildZoneList2 is the Phase 2 implementation of the **BuildZoneList** function. To get the local zone name, it makes a call to the .XPP driver with the subcode set to **zipGetZoneList**.

Listing 10-16 shows the **BuildZoneList2** routine. It begins by checking to see if a specific zone has been indicated. Line 14 checks this, and if only one zone has been specified, it is processed alone and the routine exits.

Listing 10-16. The BuildZoneList2 routine

```
PROCEDURE BuildZoneList2;
CONST
    kZonesSize      = 578;
    xCall           = 246;
    zipGetZoneList  = 6;
VAR
    theXPBPB                        : xCallParam;
    zonePtr, currZonePtr            : Ptr;
    index, count, dNode, dNet       : INTEGER;
    xppDriverRefNum                 : INTEGER;
    zoneName                        : str31;
    stat                            : OSErr;
BEGIN {BuildZoneList2}
    IF zoneParm <> '*'
        THEN BEGIN
            ProcessZone(zoneParm);
            zoneCount := 1;
            EXIT(BuildZoneList1);
        END;
    IF OpenDriver('.XPP', xppDriverRefNum) <> noErr
        THEN EXIT(BuildZoneList2);
    zonePtr := NewPtr(kZonesSize);
    IF zonePtr = NIL
        THEN EXIT(BuildZoneList2);
    WITH theXPBPB DO BEGIN
        zipInfoField[1] := 0;
        zipInfoField[2] := 0;
        zipLastFlag     := 0;
        ioRefNum    := xppDriverRefNum;
        csCode      := xCall;
        xppSubCode  := zipGetZoneList;
        xppTimeOut  := intervalNumber;
        xppRetry    := countNumber;
        zipBuffPtr  := zonePtr;
    END;
    index := 1;
    count := 0;
    SpinCursor(32);
    REPEAT
        stat := PBControl(@theXPBPB, kSYNC);
        IF stat <> noErr THEN Leave;
        count := count + theXPBPB.zipNumZones;
```

```
43:          currZonePtr := zonePtr;
44:          REPEAT
45:              zoneName :=
46:                  StrFromPtr(currZonePtr,currZonePtr^);
47:              IF (zoneName = zoneParm) OR (zoneParm = '*')
48:                  THEN BEGIN
49:                      ProcessZone(zoneName);
50:                  END;
51:              currZonePtr := Ptr(LONGINT(currZonePtr) +
52:                                      currZonePtr^+1);
53:              zoneCount := zoneCount + 1;
54:              index := index + 1;
55:          UNTIL index > count;
56:      UNTIL (theXPBPB.zipLastFlag <> 0);
57:      IF zonePtr <> NIL THEN
58:          DisposPtr(zonePtr);
59: END; {BuildZoneList2}
```

The .XPP driver is opened on line 20. If any error is encountered, it exits.

Lines 22–24 attempt to allocate a buffer for response data from the router. It's large enough to contain an entire response packet. If the call to **NewPtr** fails, the routine bails out by simply exiting.

Lines 25–35 fill in the **XPP** parameter block. Note that the **xppSubCode** field is filled in with **zipGetZoneList** to indicate to the driver which action should really be performed.

Lines 36 and 37 set up the loop. Line 38 calls **SpinCursor** to yield some time to other processes and to spin the MPW indicator, providing feedback to the user that work is being done.

Line 40 makes the call to the .XPP driver with any errors causing the loop to be exited on line 42.

Line 42 increments the zone count by the number of zones that came in the current response packet and line 43 sets the **currZonePtr** to point to the beginning of the zone names buffer.

The loop that goes from lines 56–70 processes the zones in the response packet.

Line 45 copies the zone name found at the current pointer position into **zoneName** and this is fed into the **ProcessZone** routine at line 49 as long as it either matches the specific zone requested or no specific zone was specified.

Lines 51–54 do the loop maintenance. The **currZonePtr** is bumped to point to the next zone name and the zone index is incremented to keep track of how many zone names have been processed.

Line 55 allows the inner loop to exit when all the zone names in the response packet have been processed.

Line 56 allows the outer loop to exit when the **zipLastFlag** in the XPP parameter block is TRUE.

Lines 57 and 58 clean up by deallocating the memory used to store the response packet.

ProcessZone

ProcessZone does the NBP name lookup for the specified zone. It calls **PLookupName** and then processes the buffer that is returned so it can display the name and other information of each matching name.

Listing 10-17 shows the **ProcessZone** routine. It begins by trying to allocate a large buffer at line 13 that will contain all the raw data from the **PLookupName** routine. If it can't be allocated, lines 13 and 14 write out an error message and exit.

Listing 10-17. The ProcessZone routine

```
 1: PROCEDURE ProcessZone(zoneName : str255);
 2: CONST
 3:     kBigBuffSize = 10000;
 4: VAR
 5:     EntityBuff   : str255;
 6:     bigBuff      : Ptr;
 7:     i            : integer;
 8:     addr         : AddrBlock;
 9:     anEntity     : EntityName;
10:     theMPPPB     : MPPParamBlock;
11:     stat         : OSErr;
12: BEGIN {ProcessZone}
13:     bigBuff := NewPtr(10000);
14:     IF bigBuff = NIL
15:         THEN BEGIN
16:             Writeln('bigBuff = NIL');
17:             Exit(ProcessZone);
18:         END;
19:     NBPSetEntity(@EntityBuff,nameParm,typeParm,zoneName);
20:     WITH theMPPPB DO BEGIN
```

```
21:         EntityPtr   := @EntityBuff;
22:         retBuffPtr  := bigBuff;
23:         retBuffSize := kBigBuffSize;
24:         Interval    := intervalNumber;
25:         count       := countNumber;
26:         maxToGet    := 1000;
27:     END;
28:     stat := PLookUpName(@theMPPPB,kSYNC);
29:     IF stat = noErr
30:         THEN BEGIN
31:             FOR i := 1 TO theMPPPB.numGotten DO BEGIN
32:                 nameCount := nameCount + 1;
33:                 stat := NBPExtract(bigBuff,
34:                     theMPPPB.numGotten,i,anEntity,addr);
35:                 write('"',anEntity.objStr,'" ');
36:                 write('"',anEntity.typeStr,'" ');
37:                 write('"',zoneName,'" ');
38:                 WITH addr DO BEGIN
39:                     writeln(' net=',aNet:1,' node=',
40:                         aNode:1,' socket=',aSocket:1);
41:                 END;
42:                 PLFlush(output);
43:                 SpinCursor(32);
44:             END
45:         END
46:         ELSE BEGIN
47:             CheckStat(stat,'PLookUpName');
48:         END;
49:     IF bigBuff = NIL
50:         THEN BEGIN
51:             DisposPtr(bigBuff);
52:         END;
53: END; {ProcessZone}
```

Line 19 sets up the NBP entity that describes the NBP name that should be looked up. The values either come from the MPW command line, or they are defaults.

Lines 20–27 fill in the MPP parameter block for the **PLookupName** call at line 28.

If the **PLookupName** call is error free, lines 30–45 process the results. Otherwise, line 47 logs the error using the **CheckStat** routine.

Line 31 sets up a FOR loop that repeats for each name that has been retrieved.

Line 32 increments the counter for the number of names that have been found during all processing.

Line 33 calls **NBPExtract** to retrieve the data for the specified name in the buffer.

Lines 35–41 format the resulting data and write it to the MPW standard output. Line 42 forces the output to be displayed immediately. Without this line, MPW would buffer the output.

Lines 49–52 deallocate the buffer that was used to hold the results of the **PLookupName** call.

▶ The Main Routine

With all of the routines (previously discussed in this chapter) available, the main routine for **NameTool** is fairly short. It begins by parsing the command line, then it does the ZIP and NBP processing, and finishes by writing out some statistics.

Listing 10-18 shows the main routine. It begins with a call to **InitCursorCtl**. This sets up the cursor spinning that MPW tools use to indicate to the user that progress is being made.

Listing 10-18. The main routine

```
BEGIN {main}
    InitCursorCtl(NIL);
    IF GetParm('m',zoneParm)
        THEN BEGIN
            zoneParm := GetMyZone1;
        END
        ELSE
            IF NOT GetParm('z',zoneParm)
                THEN BEGIN
                    zoneParm    := '*';
                END;
    IF NOT GetParm('n',nameParm)
        THEN BEGIN
            nameParm := '=';
        END;
    IF NOT GetParm('t',typeParm)
        THEN BEGIN
            typeParm := '=';
        END;
```

```
20:     IF GetParm('i',tempStr)
21:         THEN BEGIN
22:             StringToNum(tempStr,intervalNumber);
23:         END
24:         ELSE BEGIN
25:             intervalNumber := 3;
26:         END;
27:     IF GetParm('c',tempStr)
28:         THEN BEGIN
29:             StringToNum(tempStr,countNumber);
30:         END
31:         ELSE BEGIN
32:             countNumber := 3;
33:         END;
34:     IF GetParm('h',tempStr)
35:         THEN BEGIN
36:             PrintHelp;
37:         END
38:         ELSE BEGIN
39:             CheckStat(MPPOpen,'MPPOpen');
40:             zoneCount := 0;
41:             nameCount := 0;
42:             BuildZoneList;
43:             writeln('------------');;
44:             writeln(zoneCount:2,' zones');
45:             writeln(nameCount:2,' NBP names');
46:         END;
47: END.
```

Next it checks for the existence of the local zone option at line 3. If it's found, the **zoneParm** is set to be the local zone name. Otherwise it checks for the z option, setting the **zoneParm** to its value or to the default '*'.

Line 12 checks for the existence of the name option setting **nameParm** to its value or the default '='.

Line 16 checks for the existence of the type option setting **typeParm** to its value or the default 3.

Line 20 checks for the existence of the interval option setting **intervalNumber** to its value or the default '='.

Line 27 checks for the existence of the count option, setting **countNumber** to its value or the default 3.

Line 34 checks for the existence of the help option. If it's found the help text is written out. Otherwise normal processing will take place.

Line 39 opens the .MPP driver and lines 40 and 41 zero out the statistical counters.

Line 42 calls the real processing routine.

Lines 43–45 write out some statistics about what was found during the processing routines.

▶ Summary

This chapter described the **NameTool** MPW tool. It explored the use of the Name Binding Protocol and the Zone Information Protocol in Phase 1 and Phase 2 AppleTalk networks. Using the AppleTalk Transaction Protocol to communicate with a local router was also covered in this chapter.

Chapter 11 discusses **RemoteSysInfo**, providing an example of a Chooser RDEV/INIT.

11 ► RemoteSysInfo: An RDEV/INIT Example Program

RemoteSysInfo is an RDEV/INIT that allows you to retrieve system information about another Macintosh over an AppleTalk network. It is made up of two parts: The RDEV (a Chooser interface) and an INIT that installs an ATP socket listener in the system heap.

The RDEV part allows the user to browse through all the Macintoshes on the network that have **RemoteSysInfo** installed. It lets the user connect to any of the remote Macintoshes and displays some system information about that remote machine.

The INIT part installs a code resource that responds to the requests made by the RDEV part. It operates completely asynchronously using a state machine model.

► Goals for RemoteSysInfo

RemoteSysInfo illustrates using ATP to exchange information on an AppleTalk network. It also explores the use of Chooser interfaces, and an INIT that installs some resident code into the system heap, which asynchronously listens for requests coming into an ATP socket. **RemoteSysInfo** attempts to provide a realistic look at how this type of utility program is built without getting bogged down with too much complexity. The task it performs—getting system information—may not be too terribly exciting. However, **RemoteSysInfo** does provide a good teaching example, and can serve as the base for implementing a more complex and challenging utility.

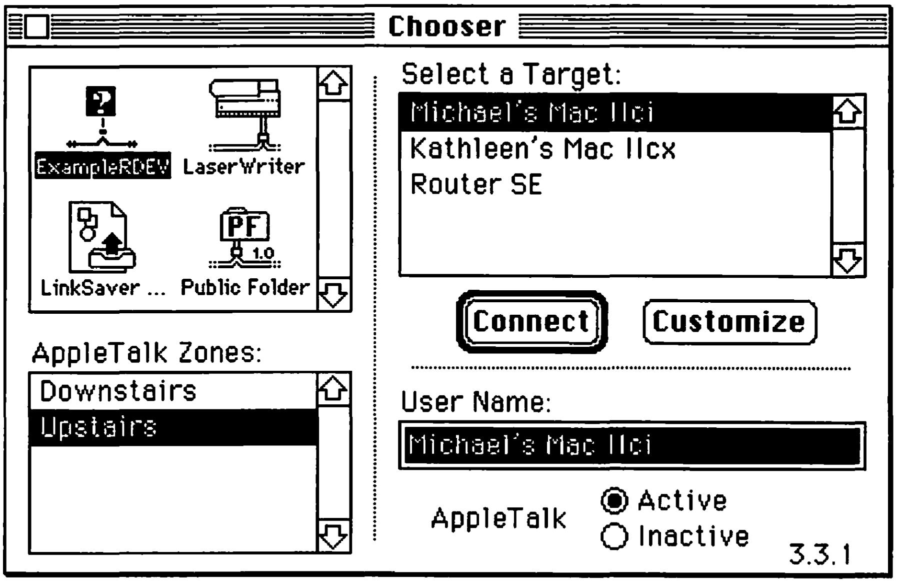
▶ How to Use RemoteSysInfo

To use **RemoteSysInfo**, drop it into your System folder and reboot. If no errors are encountered, it displays its icon during system startup. If any error is encountered (such as AppleTalk being turned off), it displays an icon with an X drawn through it.

Once **RemoteSysInfo** has been installed on at least one Macintosh in your network, you can use the Chooser interface. This is done by selecting the Chooser from the Apple Menu, and selecting the **RemoteSysInfo** icon. The Chooser will then display all the Macintoshes that have **RemoteSysInfo** installed.

You select the Macintosh that you would like to inspect by clicking on its name. Pressing the connect button will bring up a dialog of that machine's system information.

Figure 11-1 shows what the Chooser looks like when the **RemoteSysInfo** icon is selected. It lists three Macintoshes running **RemoteSysInfo**.



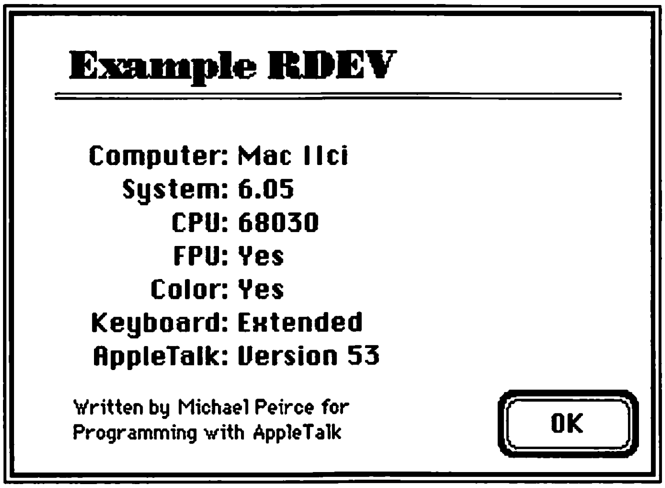
Select a Target:
Michael's Mac IIci
Kathleen's Mac IIcx
Router SE

AppleTalk Zones:
Downstairs
Upstairs

User Name: Michael's Mac IIci

AppleTalk [✓ Active] Inactive

Figure 11-1. RemoteSysInfo selected in the Chooser


Example RDEV

Computer: Mac IIci
System: 6.05
CPU: 68030
FPU: Yes
Color: Yes
Keyboard: Extended
AppleTalk: Version 53

Written by Michael Peirce for
Programming with AppleTalk

Figure 11-2. RemoteSysInfo displaying the remote system information

Figure 11-2 shows the modal dialog that **RemoteSysInfo** displays once a particular remote Macintosh is selected. It is displayed after the remote Macintosh is sent an ATP request and it sends back its response.

▶ The Structure of RemoteSysInfo

RemoteSysInfo is made up of three code resources that are executed at different times. They are: INIT, ExRC, and RDEV. There is an INIT code resource that is run at system startup time. It loads a resident code resource into the system heap. This resident code resource handles the incoming requests from the network. The RDEV code resource is executed when the Chooser brings it in. Figure 11-3 shows the three code resources in the **RemoteSysInfo** file. It illustrates the three code resources that make up this example and how the ExRC code exchanges data with the RDEV code over AppleTalk through their respective ATP sockets.

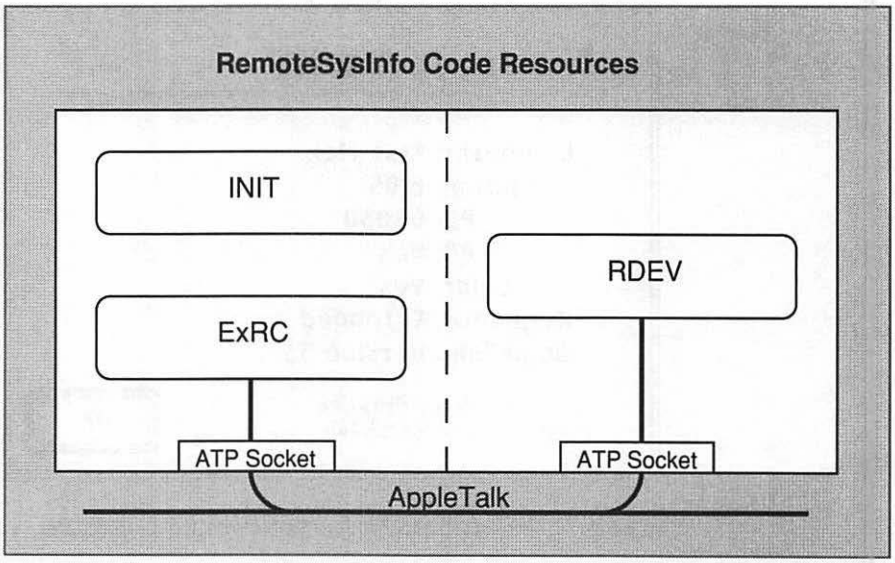


Figure 11-3. The RemoteSysInfo code resources

▶ The Source Code Structure of RemoteSysInfo

A number of source files are used to generate the three code resources that make up **RemoteSysInfo**. Figure 11-4 shows how these source files are combined to produce the three code resources.

The first code resource, the INIT, is made up of three source files.

- **ExampleInit.p**—the main code of the INIT
- **ShowInit.a**—the utility unit that allows INITs to display their icon at startup time
- **ExampleCommonUnit.p**—the unit that contains all the common definitions for **RemoteSysInfo**

The second code resource, the RDEV, is also made up of three source files.

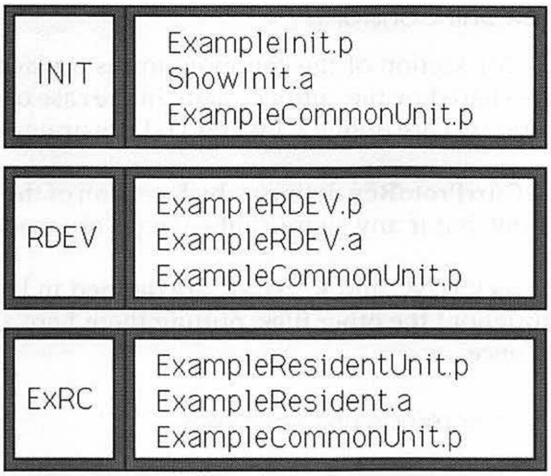


Figure 11-4. Source code files that make up each code resource

- **ExampleRDEV.p**—the main code of the RDEV
- **ExampleRDEV.a**—the assembly language code that defines the RDEV header
- **ExampleCommonUnit.p**—the unit that contains all the common definitions for **RemoteSysInfo**

The third code resource, the ExRC (Example Resident Code), is also made up of three source files.

- **ExampleResidentUnit.p**—the main code of the ExRC
- **ExampleResident.a**—the assembly language code that provides the glue for the completion routines
- **ExampleCommonUnit.p**—the unit that contains all the common definitions for **RemoteSysInfo**

▶ The Common Unit

ExampleCommonUnit.p provides a place to put all the definitions that are shared by the various parts of **RemoteSysInfo**. All other code references this unit, ensuring that they will each be using the same definitions. It doesn't contain any code, only constants and type definitions.

The Common Unit Constants

The CONSTANT section of the common unit is a place to put any constants that are shared by the entire system. In the case of **RemoteSysInfo,** only three constants are defined. Listing 11-1 illustrates the common unit constants.

On line 2, **kCurrProtoRev** defines which version of the protocol is being used. It's 1 now, but if any significant changes are made, this should be bumped up.

The constants kSYNC and kASYNC are defined in lines 3 and 4. They are used throughout the other files; putting them here allows them to be defined only once.

Listing 11-1. The common unit constants

```
1: CONST
2:   kCurrProtoRev = 1;
3:   kSYNC         = FALSE;
4:   kASYNC        = TRUE;
```

The Common Unit Types

The TYPE section of the common unit is a place to put any types that are shared by the entire system. In the case of **RemoteSysInfo,** these types fall into three broad categories, based on how the types are used. Common unit types are:

- used by the ATP messages
- used to define parameter blocks with additional space added for storing a pointer to the common variables record
- used to define the common variables record

Listing 11-2 shows the types used to define the messages sent over ATP. It begins by defining the packet types on lines 1 and 2. The enumerated type lists each type of packet that can be sent across the network. This would be added to as new packet types are defined.

Listing 11-2. Message related types

```
 1:   PacketType = (kACK,kNAK,kSysEnvironsReq,
 2:     kSysEnvironsResp);
 3:
 4:   SysEnvironsRespRec = RECORD
 5:     status       : OSErr;
 6:     SysEnvData   : SysEnvRec;
 7:   END; {ProbeReqRec}
 8:
 9:   PacketRec = RECORD
10:     version : longint;
11:     case kind : PacketType of
12:       kACK,kNAK          : (status    : integer);
13:       kSysEnvironsReq    : (filler1   : str32);
14:       kSysEnvironsResp   : (
15:                 sysEnvRespRec : SysEnvironsRespRec);
16:   END; {PacketRec}
```

Lines 4–7 define a type that is used to send the **SysEnvirons** data back in response to a request. It contains a field that will contain the status returned from the **SysEnvirons** call and a field that will contain that **SysEnvirons** data block itself.

Finally, lines 9–16 define the packet record itself. The packet record is a variant record, and takes on different forms, based on which kind of packet is being sent. It contains a version field that is always filled in with the protocol version that the code is using, which is defined in a constant in this unit. Line 11 shows a tag field for this variant record. It determines which variation of the record is used.

Line 12 defines the variant part of the packet record for the kACK and kNAK packets. These are simple packets that only contain a status value.

Line 13 defines the **SysEnvirons** request packet. It contains no real data, so a filler field is used since at least one field is required in this construct.

Lines 14 and 15 define the **SysEnvirons** response packet, which uses the **SysEnvironsRespRec** defined in line 4.

The next types defined in the common unit are those used to store a pointer ahead of a parameter block. This is used to gain access to the block at interrupt time. See Chapter 3 for a discussion of this technique.

Listing 11-3 shows an extended parameter block for the ATP parameter block. In this example, only one extended parameter block is used.

Listing 11-3. The ATP extended parameter block

```
myATPParamBlock = RECORD
  dataPtr      : LONGINT;
  realATPPB    : ATPParamBlock;
END; {myATPParamBlock}
```

The last type defined in the common unit contains all the variables that need to be preserved across calls to the completion routine. Listing 11-4 shows this.

Listing 11-4. The completion routine data block

```
dataBlockRec = RECORD
  stat              : OSErr;
  sig               : str31;
  outATPPB,
  inATPPB           : myATPParamBlock;
  mySocket          : byte;
  inPacket          : PacketRec;
  responsePacket    : PacketRec;
  outBDS            : BDSType;
  entityBuff        : str255;
END; {dataBlockRec}

DataBlockPtr = ^DataBlockRec;
```

Listing 11-4 contains the various parameter blocks and other buffers that are needed by the resident code. Note the **sig** field in the **dataBlockRec**. This is filled in with a short string unique to your program. Then as you are debugging your code, you can have your debugger search for this string to locate the block in the system heap. It can also be useful as a sanity check when viewing the block to make sure it really is your block.

▶ The Example INIT

ExampleINIT.p is the place where all the startup code for **RemoteSysInfo** goes. It makes use of two other units: **ShowInit** and **ExampleCommonUnit**. **ShowInit** is a library function, available from Apple, that provides a way for an INIT to display a startup icon. **ExampleCommonUnit.p** provides a number of common elements for the **RemoteSysInfo** system and was explained earlier.

ExampleINIT is organized as three nested routines. This is done to divide them into three easily understood pieces of routines. **INITEntry**, an outer routine, handles drawing the icons and calling the next level, **ExampleResidentCodeInstall. ExampleResidentCodeInstall** opens the AppleTalk .MPP driver and installs the resident code. Then it calls the next level, **GoOnline**, which registers an NBP name and calls **PGetRequest** with the resident code providing the completion routine.

INITEntry

INITEntry is a fairly simple routine. Its purpose is to display the startup icons to tell the user if the INIT has been installed properly or not. The icon with the X through it is displayed if the user tells it not to load by holding down either the Shift key or the mouse button. The X icon will also be displayed if any error is encountered while installing the resident code. **INITEntry** is shown in Listing 11-5.

Listing 11-5. INITEntry

```
PROCEDURE INITEntry;
CONST
    kNormalICON = -4080;
    kErrorIcon  = -4079;
    shiftKey    = 56;
VAR
    theKeys         : KeyMap;
    aLong,lastTick  : longint;
    iconID          : integer;
BEGIN
    GetKeys(theKeys);
    IF Button  or theKeys[shiftKey]
        THEN BEGIN
            showINIT(kErrorIcon,-1);
        END
        ELSE
            IF ExampleResidentCodeInstall = noErr
                THEN BEGIN
                    showINIT(kNormalICON,-1);
                END
                ELSE BEGIN
                    showINIT(kErrorIcon,-1);
                END;
END;
```

INITEntry begins by checking for the Shift key or mouse button being down on lines 11-12. These are generally accepted ways to tell an INIT not to load.

If the Shift key or the mouse button is down, the error icon is shown on line 14 and no further processing is done. Otherwise line 17 calls the routine that attempts to install the resident code. If this succeeds the standard icon is displayed on line 19. If the resident code is not successfully installed, the error icon is displayed on line 22.

ExampleResidentCodeInstall

ExampleResidentCodeInstall does much of the work of the INIT. It opens the AppleTalk .MPP driver, turns on **SelfSend**, loads in the resident code resource, allocates the global data block in the system heap, then calls the **GoOnline** routine. **ExampleResidentCodeInstall** is shown in Listing 11-6.

Listing 11-6. ExampleResidentCodeInstall

```
 1: FUNCTION ExampleResidentCodeInstall : OSErr;
 2: VAR
 3:     DataBlockP   : DataBlockPtr;
 4:     stat             : osErr;
 5:     serverName       : str255;
 6:     userName         : stringHandle;
 7:     theMPPPB         : MPPParamBlock;
 8:     theATPPB         : ATPParamBlock;
 9:     refNum           : integer;
10:     codeH            : handle;
11: {GoOnline is nested here}
12: BEGIN {ExampleResidentCodeInstall}
13:     stat := OpenDriver('.MPP',refNum);
14:     IF stat = noErr
15:         THEN BEGIN { go ahead }
16:             theMPPPB.newSelfFlag := 1;
17:             stat := PSetSelfSend(@theMPPPB,kSYNC);
18:             SetZone(SystemZone);
19:             codeH := Get1Resource('ExRC',1);
20:             IF codeH = NIL
21:                 THEN BEGIN
22:                     sysbeep(5);
23:                     stat := 5;
24:                     DebugStr('No ExRC resource!');
25:                 END
```

```
26:                 ELSE BEGIN
27:                     DetachResource(codeH);
28:                     HNoPurge(codeH);
29:                     HLock(codeH);
30:                     DataBlockP := DataBlockPtr(
31:                         NewPtr(SIZEOF(DataBlockRec)));
32:                     SetZone(ApplicZone);
33:                     DataBlockP^.sig :=
34:                         'RemoteSysInfo datablock    ';
35:                     GoOnline;
36:                     ExampleResidentCodeInstall := noErr;
37:                 END;
38:         END {noErr}
39:         ELSE BEGIN
40:             ExampleResidentCodeInstall := stat;
41:         END;
42: END; {ExampleResidentCodeInstall}
```

GoOnline begins by opening the .MPP driver on line 13. If this fails, it does nothing further and just returns this status to its caller.

If the .MPP driver is successfully opened, lines 16 and 17 attempt to set **SelfSend** to be on. Any error is ignored.

Line 18 sets the current heap zone to be the system heap zone. This allows line 19 to load the resident code resource into the system heap. It must be loaded there because it will be left there after the INIT terminates.

Line 20 checks to make sure the resource loaded. If it did not, it makes some noise and calls the debugger to tell you about the problem in lines 22–24. This type of code is very helpful when developing your code, but should be removed before you release it!

Line 27 detaches the code resource from the resource file. If you fail to do this, the resource can be purged when the resource file is closed.

Line 28 and 29 ensure that the handle (no longer a resource) will not be moved or removed.

Line 30 allocates the global data storage the resident code needs. Because the current heap is still the system heap, it will reside there.

Line 32 sets the current heap to be the application heap so that any further resources will load there.

Line 33 puts a signature into the data block. This can help you to find the data block in memory later during debugging.

Line 35 calls the **GoOnline** routine.

If the code has reached line 36 without error, it sets the return code for the routine to **noErr**.

GoOnline

GoOnline does three basic tasks: it opens an ATP socket, assigns a unique name to it, and issues an asynchronous **PGetRequest** using the socket. The **GoOnline** routine is shown in Listing 11-7.

Listing 11-7. The GoOnline routine

```
PROCEDURE GoOnline;
VAR
    nameCount   : integer;
    theNBPType  : str255;
    countStr    : str255;
BEGIN {GoOnline}
    WITH DataBlockP^ DO BEGIN
        WITH theATPPB DO BEGIN
            atpsocket := 0; {ask for any available}
            addrBlock.aNet      := 0;
            addrBlock.aNode     := 0;
            addrBlock.aSocket   := 0;
        END;
        IF POpenATPSkt(@theATPPB,kSYNC) <> noErr
            THEN DebugStr('POpenATPSkt');
        mySocket := theATPPB.atpSocket;
        userName   := GetString(-16096);
        theNBPType := GetString(-4096)^^;
        serverName  := userName^^;
        nameCount   := 1;
        REPEAT
            NBPSetNTE(@entityBuff,serverName,
                    theNBPType,'*',mySocket);
            WITH theMPPPB DO BEGIN
                EntityPtr   := @entityBuff;
                Interval        := 3;
                count           := 3;
                verifyFlag      := $FF;
            END;
            stat := PRegisterName(@theMPPPB,kSYNC);
            IF stat = nbpDuplicate
                THEN BEGIN
                    nameCount := nameCount + 1;
                    HLock(Handle(userName));
                    NumToString(nameCount,countStr);
                    serverName := concat(userName^^,
                                    ' [',countStr,']');
                    HUnlock(Handle(userName));
                END;
```

```
40:         UNTIL stat <> nbpDuplicate;
41:         IF stat <> noErr
42:             THEN BEGIN
43:                 DebugStr('PRegisterName');
44:             END;
45:         WITH inATPPB.realATPPB DO BEGIN
46:             atpSocket   := mySocket;
47:             reqLength   := SIZEOF(PacketRec);
48:             reqPointer := @inPacket;
49:             numOfBuffs := 1;
50:             ioCompletion := codeH^;
51:         END;
52:         inATPPB.dataPtr := LongInt(DataBlockP);
53:         IF PGetRequest(@inATPPB.realATPPB,kASYNC) <> noErr
54:             THEN DebugStr('PGetRequest');
55:     END;
56: END; {GoOnline}
42: END; {ExampleResidentCodeInstall}
```

GoOnline begins by opening an ATP socket at line 14. Lines 9–12 specify that AppleTalk should give it any open socket and that requests should be accepted from any network address.

Line 16 stores the returned socket number into the data block for use later.

Lines 17 and 18 get the user's Chooser name, stored in the System file as string ID = –16096, and the NBP type string, stored in the program file as string ID = –4096.

Line 19 makes a copy of the server name so that it can be used later to generate a unique name if need be.

The loop beginning at line 21 is used to repeatedly generate server names and test for their uniqueness. If a name isn't unique, the string "[n]" is appended to the user name, with n being a number, until a unique name is found. This generates a progression of names such as "Michael", "Michael [1]", "Michael [2]", and so on.

Line 22 creates the names table entry for the desired name.

Lines 24–29 fill in the parameter block with appropriate values for the **NBPRegisterName** call found at line 30.

If the **NBPRegisterName** call returns **nbpDuplicate** (indicating that the desired name is already in use on the network), lines 33–38 generate the next possible name in the progression.

Line 40 terminates the loop when no duplicate names are found.

Of course other errors besides duplicate names can be returned from **NBPRegisterName**. If this happens, lines 41–44 report this by calling the debugger.

Lines 45–51 fill in the ATP parameter block with values appropriate for the **PGetRequest** call. Line 50 sets the completion routine to be the code starting at the beginning of the resident code resource, assuming that the first instructions found in the code resource implement the completion routine.

Line 52 stores a pointer to the global data block into the data pointer space found right before the ATP parameter block. This allows the completion routine to easily find the data block at interrupt time.

Line 53 calls **PGetRequest** asynchronously. Note that the code resource must have been loaded in and locked down prior to this call.

▶ The Resident Code

The code that makes up the resident code resource lives in two files

- **ExampleResident.a**—contains the assembly language glue code
- **ExampleResidentUnit.p**—contains the Pascal code

The resident code also references the **ExampleCommonUnit.**

The resident code is called only at interrupt time in response to the completion of the **PGetRequest** call. It processes the incoming request and sends back a response. It must then issue another **PGetRequest** with itself as the completion routine so that it will be called again when another request comes in.

The Resident Assembly Routine

The resident assembly code has two functions. Its first function is to save the processor registers upon entering the routine and to restore them to their previous state before leaving. Its second function is to fetch the pointer to the global data block stored ahead of the parameter block and pass it in as a parameter to the Pascal completion routine. Listing 11-8 illustrates the resident assembly routine.

Listing 11-8. The resident assembly routine

```
 1: ;---------------------------------------------------------
 2: ;
 3: ;   Example Resident Code Macro Glue
 4: ;
 5: ;   The idea here is that the completion routines need
 6: ;   to have some context (the data block where we store
 7: ;   all our variables) when they run.  So what we do
 8: ;   here in front of the io block, and push it onto the
 9: ;   stack.  This allows the Pascal code to take it in
10: ;   as a parameter and then do a nice big WITH.  The
11: ;   effect is that all data is stored off A4 (setup by
12: ;   the with) instead of A5 (bad carma is used from
13: ;   within a completion routine.
14: ;
15: ;   Oh yes, we also do the usual register store/reload
16: ;   stuff here so we don't step on interrupted code.
17: ;
18:     BLANKS      ON
19:     INCLUDE     '::AIncludes:SysEqu.a'
20:     IMPORT      INCOMING
21: ;---------------------------------------------------------
22: XINCOMING PROC      EXPORT  ; Glue for INCOMING procedure
23:
24:     MOVEM.L     D3-D7/A2-A6,-(A7) ; save them registers
25:     MOVE.L      -4(A0),D0         ; get our block pointer
26:     MOVE.L      D0,-(A7)          ; pass it as a parameter
27:     JSR         INCOMING          ; call Pascal routine
28:     MOVEM.L     (A7)+,A2-A6/D3-D7 ; restore the registers
29:     RTS
30: ;---------------------------------------------------------
31:     END
```

The resident assembly routine begins by saving the processor registers at line 24. Note that registers D0, D1, D2, A0, and A1 need not be preserved because the interrupt dispatcher does this for you.

Lines 25 and 26 retrieve the global data block pointer and push it onto the stack for the Pascal routine. Note that completion routines are called with A0 pointing to the ATP parameter block. This allows you to access the global data block pointer with a negative 4 byte offset from A0.

Line 27 calls the Pascal completion routine.

Line 28 restores the processor registers to the state they were in before executing the completion routine.

Line 29 returns from this subroutine.

Incoming

Incoming is the code that processes the incoming ATP request. It handles the request by looking at the packet kind and handling each kind separately. After processing each request, it then sends back a response and makes a call to **PGetRequest** to be ready for the next request. Listing 11-9 illustrates the **Incoming** routine.

Listing 11-9. Incoming

```
PROCEDURE Incoming(dataBlock : DataBlockPtr);
BEGIN {Incoming}
    WITH dataBlock^ DO BEGIN
        outATPPB := inATPPB;
        WITH outATPPB.realATPPB DO BEGIN
            atpSocket       := mySocket;
            atpFlags        := atpEOMvalue;
            {addrBlock      := …from the incoming Packet}
            bdsPointer      := @outBDS;
            bdsSize         := 1;
            numOfBuffs      := 1;
            {transID        := …from the incoming Packet}
            ioCompletion    := NIL;
        END;
        responsePacket.version := kCurrProtoRev;
        CASE inPacket.kind OF
            kSysEnvironsReq: BEGIN
                WITH responsePacket DO BEGIN
                    kind := kSysEnvironsResp;
                    sysEnvRespRec.status :=
                        SysEnvirons(1,
                            sysEnvRespRec.SysEnvData);
                END;
            END;
            otherwise BEGIN
                responsePacket.kind     := kNAK;
                responsePacket.status   := noErr;
            END;
        END; {CASE}
        SendResponse(dataBlock);
    END;
END; {Incoming}
```

Listing 11-10. SendResponse

```
 1: PROCEDURE SendResponse(var dataBlock : DataBlockPtr);
 2: VAR
 3:     buffSize    : INTEGER;
 4: BEGIN {SendResponse}
 5:     WITH dataBlock^ DO BEGIN
 6:         CASE responsePacket.kind OF
 7:             kACK,
 8:             kNAK            : buffSize := 8 + 2;
 9:             kSysEnvironsReq,- not generated by server}
10:             kSysEnvironsResp: buffSize := 8 +
11:                                SIZEOF(SysEnvironsRespRec);
12:             otherwise         buffSize :=
                                    SIZEOF(PacketRec);
13:         END;
14:         WITH outATPPB.realATPPB DO BEGIN
15:             bdsSize := BuildBDS(@responsePacket,
16:                                 @outBDS,buffSize);
17:             numOfBuffs := bdsSize;
18:         END;
19:         stat := PSendResponse(@outATPPB.realATPPB,kASYNC);
20:         IF stat <> noErr THEN BEGIN
21:             debugStr('pSENDResp');
22:         END;
23:         WITH inATPPB.realATPPB DO BEGIN
24:             reqLength       := SIZEOF(PacketRec);
25:             atpSocket       := mySocket;
26:             reqPointer      := @inPacket;
27:             ioCompletion    := @XIncoming;
28:         END;
29:         stat := PGetRequest(@inATPPB.realATPPB,kASYNC);
30:         IF stat <> noErr THEN BEGIN
31:             debugStr('pGetReq');
32:         END;
33:     END;
34: END; {SendResponse}
```

SendResponse begins by using the data block pointer in a WITH statement on line 5. This allows any of the fields in the data block to be accessed without explicitly referencing the pointer.

Incoming begins by using the data block pointer passed in a WITH statement on line 3. This allows any of the fields in the data block to be accessed without explicitly referencing them off the pointer.

At line 4 the parameter block used by the **PGetRequest** is copied into the parameter block used by the **PSendResponse**. (The **PSendResponse** call shown in Listing 11-10 copies a number of important fields such as the address of the requester and the transaction ID.)

Lines 5–14 fill in the ATP parameter block used by the **PSendResponse** with appropriate values.

Line 15 copies the protocol revision level that this code was built with into the version field of the response message. This can be used on the other end to deal with new revisions to the packet formats.

At line 16 is the case statement that handles each of the possible incoming request packet types. In this example, there aren't many variations, but this is where you would expand to handle new packet types.

Lines 17–24 handle the requests for system information. Line 19 sets the response packet's kind to be a system information response and lines 20–22 fill in the **SysEnvData** field with the results of a call to **SysEnvirons**.

Lines 25–29 handle any other incoming requests. Because the code isn't built to handle any other type of request, it responds to other types of requests with a kNAK. This tells the requester that its request wasn't understood. This is handy for later when an enhanced version of the code hits the network. It's very hard to get rid of old versions, and this allows those old versions to behave in a reasonable way when they get requests they don't understand.

Finally, line 30 calls the **SendResponse** routine to ship the response back to the responder and reissue the **PGetRequest** call. It also passes the data block pointer to the **SendResponse** routine so that it can access the data stored there.

SendResponse

SendResponse is the code that sends the response message back to the requester. It figures the correct message size based on the type of message being sent. It also reissues the **PGetRequest** so that additional requests can be received later. Listing 11-10 shows the **SendResponse** routine.

The case statement at lines 6–13 figures the correct message size to send. Each message kind has a separate case that does the right calculation for that message kind. With the few message types used by this example, there isn't much variation here; if you want to handle many types of messages, more cases get added here. Note that the calculation adds 8 bytes (the version, kind, and some slop) to the size of the record used by each kind of response.

Line 19 sends the response back to the requester.

Lines 23–28 prepare the ATP parameter block for issuing another **PGetRequest** at line 29. Note that line 27 sets the completion routine to be the **Xincoming** assembly language routine.

Finally the **PGetRequest** is called asynchronously at line 29.

▶ The RDEV Code

The code that makes up the RDEV code resource lives in two files: **ExampleRDEV.a** contains the assembly language that provides the PACK header that begins an RDEV, and **ExampleRDEV.p** contains the Pascal code. It also references the **ExampleCommonUnit.p**.

The RDEV code resource takes the form of a PACK resource. This means that the header defined in the assembly language code must be linked in as the first part of the code resource.

This code is divided into a number of subroutines. The assembly code found in Listing 11-11 consists of just the PACK header and a jump to the main Pascal routine. The main Pascal routine handles the messages sent from the Chooser. It calls the main dialog handling routine if appropriate. The main dialog handling routine sends the request to the remote Macintosh and formats the response into the modal dialog. It also makes use of a number of utility routines.

The RDEV Assembly Routine

The RDEV assembly code has two functions. Its first function is to define the PACK structures that the Chooser looks at. The second function is to jump to the actual RDEV handling code written in Pascal. Listing 11-11 illustrates the RDEV assembly routine.

Listing 11-11. The RDEV assembly routine

```
 1:          BLANKS      ON
 2:          INCLUDE     '::AIncludes:SysEqu.a'
 3: ExampleRDEV PROC     EXPORT
 4:          IMPORT      pExampleRDEV
 5:          BRA.S       @1           ; skip over header
 6:          DC.W        80           ; Device ID
 7:          DC.L        ('PACK')     ;
 8:          DC.W        $F000        ; -4096
 9:          DC.W        1            ; version 1
10:          DC.L        %10001110000000000000000000000000
11:                      ;    ^   ^   ^   ^   ^   ^   ^
12: @1       JMP         pExampleRDEV
13:          END
```

The RDEV assembly routine begins with a branch instruction at line 5. This is needed because the Chooser not only calls the first instruction of the code resource, but also requires that data be stored immediately after that first instruction. So you simply jump over the data to code placed after the data.

Line 6 declares the device ID word. This code simply fills it with a value of 80.

Line 7 declares the constant four characters, 'P', 'A', 'C', and 'K', required in a 'PACK' code resource.

Line 8 declares the constant –4096 ($F000 in hex) required of an RDEV.

Line 9 declares the version of this RDEV, version 1.

Line 10 declares the flags used by the Chooser to determine the behavior of an RDEV. This specifies that, yes, this is an RDEV, that both right and left buttons are used, and that this RDEV uses AppleTalk. See Figure 9–2 for all the values of these flags.

Line 12 jumps to the Pascal code that handles the RDEV.

The RDEV Main Pascal Routine

The RDEV Pascal code responds to the messages sent in from the Chooser. This example only responds to the button message. In fact, the right-button message is there simply for show. When the left-button message is pressed the main dialog handling code is called. Listing 11-12 shows the **pExampleRDEV** routine.

Listing 11-12. pExampleRDEV

```
 1: FUNCTION pExampleRDEV (
 2:     message     : INTEGER;
 3:     caller      : INTEGER;
 4:     objName     : StringPtr;
 5:     zoneName    : StringPtr;
 6:     p1          : LONGINT;
 7:     p2          : ButtonP2) : OSErr;
 8: { MainRDEVDialog is nested here }
 9: BEGIN {pExampleRDEV}
10:     IF message = buttonMsg THEN BEGIN
11:         IF p2.whichButton = kLeftButtonPressed
12:             THEN BEGIN {They pressed the CONNECT button}
13:                 MainRDEVDialog;
14:             END
15:             ELSE BEGIN {They pressed the the right button,
16:                                         the CUSTOMIZE button}
17:                 SysBeep(5);
18:                 IF p2.modifiers.optionKeyIsDown
19:                     THEN SysBeep(5);
20:             END;
21:     END;
22:     pExampleRDEV := noErr;
23: END;
```

On line 10, **pExampleRDEV** begins checking for a button message. If one is received, either the main dialog handling routine is called on line 13 or line 17 calls **SysBeep.** Note that this example uses the **ButtonP2** type discussed in Chapter 9.

Line 22 always forces the return value of **noErr** indicating that no serious problems were encountered.

The RDEV Main Dialog Routine

The main dialog routine gets the target address from the list manager list. It then sends a request to the resident code at that address. When the response comes back, it puts up a dialog that displays the information found in the response message.

Listing 11-13 shows the **MainRDEVDialog** routine. Note that this routine is nested in the **pExampleRDEV** routine so it is in the scope of the **pExampleRDEV** routine, and the **MainRDEVDialog** routine can directly access the parameters of the **pExampleRDEV** routine.

Listing 11-13. MainRDEVDialog

```
PROCEDURE MainRDEVDialog;
VAR
    info            : SysEnvRec;
    theDialog       : DialogPtr;
    itemHit         : INTEGER;
    tempStr,temp2Str: str255;
    size            : INTEGER;
    theCell         : Cell;
    ATAddr          : AddrBlock;
    theATPPB        : ATPParamBlock;
    BDSCount        : INTEGER;
    outBDS          : BDSType;
    reqPacket,
    replyPacket     : PacketRec;
    itemHandle      : Handle;
    itemRect        : Rect;
    itemType        : INTEGER;
BEGIN {MainRDEVDialog}
    theCell := Point(LONGINT(0));
    IF LGetSelect(TRUE,theCell,ListHandle(p1))
        THEN {Do Nothing};
    theCell.h := 1;
    size := SIZEOF(AddrBlock);
    LGetCell(@ATAddr,size,theCell,ListHandle(p1));
    theDialog := GetNewDialog(kMainRDEVDialogID,
                            NIL, WindowPtr( - 1));
    IF theDialog = NIL
        THEN Exit(MainRDEVDialog);
    BDSCount := BuildBDS(@replyPacket,@outBDS,
                            SIZEOF(PacketRec));
    WITH reqPacket DO BEGIN
        version := kCurrProtoRev;
        kind    := kSysEnvironsReq;
    END;
    WITH theATPPB DO BEGIN
        bdsPointer      := @outBDS;
        addrBlock       := ATAddr;
        timeOutVal      := kATPTimeOutVal;
        retryCount      := kATPRetryCount;
        numOfBuffs      := BDSCount;
        atpFlags        := atpXOvalue;
        reqLength       := SIZEOF(PacketRec);
        reqPointer      := @reqPacket;
        ioCompletion    := NIL;
    END;
```

```
  IF PSendRequest(@theATPPB,kASYNC) <> noErr
    THEN DebugStr('PSendRequest');
  REPEAT
    SystemTask;
  UNTIL (theATPPB.ioResult <> 1);
  IF (theATPPB.ioResult = noErr) AND
   (replyPacket.kind = kSysEnvironsResp) THEN BEGIN
    info := replyPacket.sysEnvRespRec.SysEnvData;
    SetDialogStringItem(theDialog,kComputerItem,
     LookupString(kMachineStrID,info.machineType));
    NumToString(LONGINT(info.systemVersion DIV 256),
                                        tempStr);
    NumToString(LONGINT(info.systemVersion MOD 256),
                                        temp2Str);
    IF LENGTH(temp2Str) < 2
      THEN BEGIN
        temp2Str := CONCAT('0',temp2Str);
      END;
    tempStr := CONCAT(tempStr,'.',temp2Str);
    SetDialogStringItem(theDialog,kSystemItem
    tempStr);
    SetDialogStringItem(theDialog,kCPUItem,
     LookupString(kCPUStrID,info.processor));
    IF info.hasFPU
      THEN tempStr := 'Yes'
      ELSE tempStr := 'No';
    SetDialogStringItem(theDialog,kFPUItem,tempStr);
    IF info.hasColorQD
      THEN tempStr := 'Yes'
      ELSE tempStr := 'No';
    SetDialogStringItem(theDialog,kColorItem,tempStr);
    SetDialogStringItem(theDialog,kKeyBoardItem,
     LookupString(kKeyBoardStrID,info.keyBoardType));
    NumToString(LONGINT(info.atDrvrVersNum), tempStr);
    tempStr := CONCAT('Version ',tempStr);
    SetDialogStringItem(theDialog,
          kAppleTalkItem,tempStr);
    GetDItem(theDialog, kOKButtonOutline,
          itemType, itemHandle, itemRect);
    itemHandle := @DefaultButtonFilter;
    SetDItem(theDialog, kOKButtonOutline,
          itemType, itemHandle, itemRect);
    ModalDialog(NIL, itemHit);
  END;
  CloseDialog(theDialog);
END; {MainRDEVDialog}
```

MainRDEVDialog begins by extracting the address of the selected item in the list. It does this on lines 19–24 by first getting the selected cell on line 20, then accessing the address data stored in column 2 of the selected row—the first column is column zero. Line 24 copies the data stored there into the **ATAddr** variable.

Line 25 loads in the dialog that will be used later to display the response data.

Line 29 builds a BDS for the response message.

Lines 31–34 build the request message to be a **SysEnvirons** request.

Lines 35–45 fill in the ATP parameter block with the appropriate values to make a request of the remote Macintosh.

Line 46 makes the **PSendRequest** call asynchronously. This allows lines 48–50 to use the polling technique discussed in Chapter 3, yielding time until the response comes back.

If a **kSysEnvironsResp** is received back, lines 54–86 process the response message into the appropriate dialog items. Processing the response uses two utility routines—one that sets a specified dialog item to be a specific string, and another that takes a number and returns a string. You specify a string by using its number as an index into an 'STR#' resource. Having a number as an index is very helpful for turning numbers like 2 or 3 into strings like "Mac Plus" or "Mac SE".

Line 87 handles the dialog until the OK button is pressed. No further dialog processing is done so the dialog is closed at line 89.

▶ Summary

This chapter explored the **RemoteSysInfo** RDEV/INIT. It described the code that went into each of the three code resources that make up **RemoteSysInfo**. First, the INIT and how it was loaded in the resident code was discussed. Then the resident code and how it serviced an incoming request at interrupt time was covered. Finally, the RDEV code and how it handled interfacing with the Chooser to provide a user interface for the user was described.

Chapter 12, the final chapter of this book, outlines an ADSP example program called Checkers. It shows you how to play a game of checkers on an AppleTalk network.

12 ▶ Checkers: An ADSP Example Program

Checkers is a MacApp-based application that allows two people to play a game of checkers over an AppleTalk network. It includes code that manages the ADSP connection between two instances of the application running, exchanging game information during play.

One unit, **UAsyncOp**, provides an abstract class, **TAsyncOp**, that implements asynchronous operations. This unit can be used to implement a wide variety of asynchronous tasks using polling for completion of operations. It is used here with ADSP, but it can also be used with other protocols such as ATP or NBP. It can also be used with non-AppleTalk functions such as serial communications, file operations, or Time Manager tasks.

The primary unit implementing the ADSP connection is **UNetStuff**. It defines a subclass of **TAsyncOp** called TADSP that implements some general ADSP functionality. Another abstract class is also defined in **UNetStuff: TPlayer**. It is a subclass of TADSP, and implements communication between the two players. **TPlayer** is a superclass for the two concrete classes: **TUs** and **TThem**; these implement the differences found in the two players.

▶ Goals for Checkers

Checkers illustrates the basic function of communicating data across an ADSP connection built within a MacApp-based framework. It uses some basic techniques from object-oriented programming to simplify and generalize the ADSP connection handling. Some of the building blocks provided can be used in other circumstances for very different purposes with little change.

▶ How to Use Checkers

This version of checkers is a fairly straightforward game to play. It uses a simple game model, so that the focus is on the network coding techniques rather than on a more sophisticated game. Once the game is launched the user can either actively select a partner over the network or sit passively waiting for a connection request to come in from someone else.

Once the connection is established, play begins. Each player has control of the board in turn. Checkers are moved by clicking and dragging. The movement of the checkers is sent over the network in real-time so that as a checker is dragged across the board, the other board also shows the movement. After each move, the state of the board is sent over the connection, and control of the board is turned over to the other player.

▶ General Comments about Checkers

The basic communications strategy used by checkers is to first open a connection, then exchange play data back and forth across the connection until the game is over.

The opening of the connection is handled differently by the two types of players. When the program starts up, the user chooses to either wait passively for a connection from someone else, or to seek out a passive player and connect to them. The passive player opens a connection end in passive mode and simply waits. The active player actively seeks out the the passive player and connects to them using request mode. Once the connection is established, play begins. The active player always goes first.

The player whose turn it is clicks and drags a checker to make a move. Moves are tracked in real-time on the other player's board by sending the **StartTracking** message followed by many **Track** messages as the checker is dragged across the board. When dragging is done, the **StopTracking** message is sent. When the move is finished, the state of the board is determined by the player making the moves, and is sent to the other player using the **UpdateBoard** message. Finally, a **YourTurn** message is sent. This turns control over to the other player and the process is continued. Play is ended when the **YourTurn** message has a flag set that indicates that the game has been won by one of the players.

To provide reasonable tracking performance, the receiving program goes into a tight loop using synchronous reads. This allows for very smooth tracking of the checkers pieces. Its drawback is that it locks out user control on the receiving Macintosh while the loop is taking place. Control is returned to the user as soon as the checker tracking stops.

Another simplification found in the code is that once a game is over, the connections are simply torn down. This is done by freeing the player objects. No special code is provided to detect half-closed connections.

▶ The Structure of the Checkers Player Objects

The ADSP-related code in checkers is contained in two rather arbitrary units. The first unit, **UAsyncOp**, contains an abstract class, **TAsyncOp**, which provides a general framework for constructing asynchronous objects. The second unit, **UNetStuff**, contains the abstract classes **TADSP** (used to construct ADSP objects), and **TPlayer** (used to construct the checkers player objects). **UNetStuff** also contains two concrete classes, **TActivePlayer** and **TPassivePlayer**, which implement the two players.

The two concrete classes, **TActivePlayer** and **TPassivePlayer**, inherit most of their behavior from their superclass. They differ only in the way they make the connection, and in which one goes first when play begins. Figure 12-1 shows the class hierarchy.

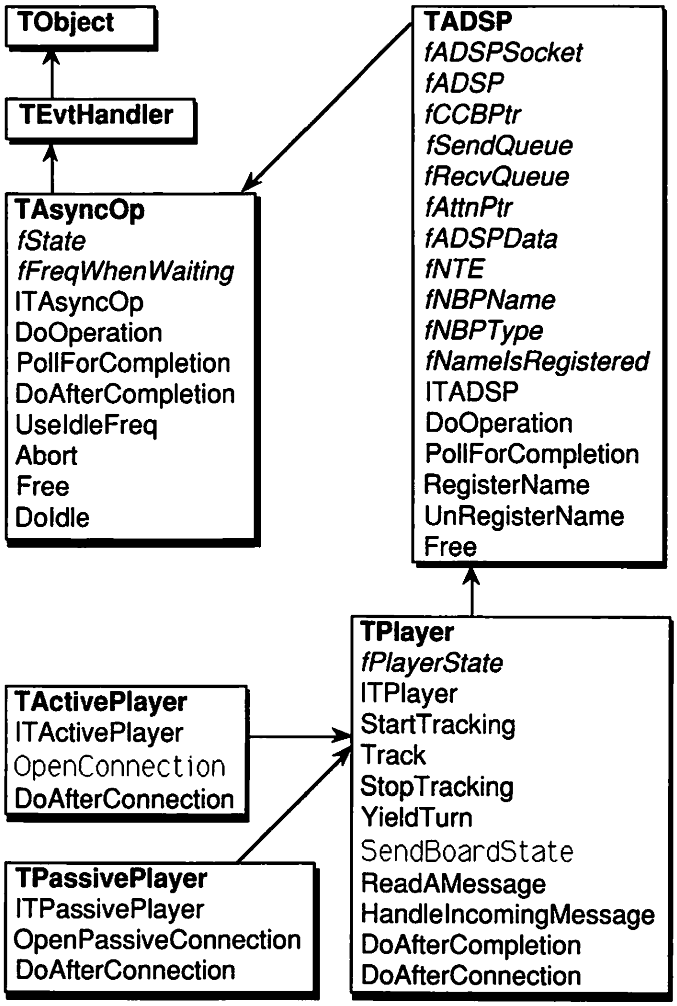


Figure 12-1. The class hierarchy

▶ Tour of the Players' Class Hierarchy

To understand how the players function and interact, you have to understand the behavior of each of the superclasses that go into making them. The following section begins with abstract class **TAsyncOp**, progresses through **TADSP** and **TPlayer**, and ends with the two concrete player classes, **TUs** and **TThem**.

▶ The TAsyncOp Class

The abstract class **TAsyncOp** is a very general abstract class. It provides a set of methods that build a framework for implementing many types of asynchronous operations.

TAsyncOp is a subclass of **TEvtHandler**. This means that it can install itself into the cohandler queue and it will be called periodically when the application has idle time. This allows **TAsyncOp** to poll for the completion of an asynchronous operation during this idle time.

The basic model **TAsyncOp** uses is that there are three basic phases to every asynchronous operation:

- setup and issuance
- polling for completion
- post-processing

Setup and issuance are performed by the subclass. These would typically be such tasks as filling in a parameter block and calling a driver asynchronously. Once this has been done, the subclass must call the **DoOperation** method to start the polling process.

Polling is performed periodically when the application has idle time. The method **PollForCompletion** is called repeatedly. It returns TRUE when the asynchronous operation has completed. This method must be overridden by the subclass to implement polling in the appropriate way.

Finally, once the asynchronous operation has completed, the **DoAfterCompletion** method is called. Again, this must be overridden by the subclass to take appropriate action. Listing 12-1 shows the declaration for the **TAsyncOp** class.

Listing 12-1. Declaration of the class TAsyncOp

```
 1:   AsyncOpStates = (kIdling,kWaiting);
 2:
 3:   TAsyncOp = OBJECT(TEvtHandler)
 4:     fState             : AsyncOpStates;
 5:     fFreqWhenWaiting   : longint;
 6:     PROCEDURE TAsyncOp.ITAsyncOp;
 7:     PROCEDURE TAsyncOp.DoOperation;
 8:     FUNCTION  TAsyncOp.PollForCompletion : BOOLEAN;
 9:     PROCEDURE TAsyncOp.DoAfterCompletion;
10:     PROCEDURE TAsyncOp.UseIdleFreq(newFreq : LONGINT);
11:     PROCEDURE TAsyncOp.Abort;
12:     PROCEDURE TAsyncOp.Free; OVERRIDE;
13:     FUNCTION TAsyncOp.DoIdle(phase: IdlePhase) :
14:         BOOLEAN; OVERRIDE;
15:     PROCEDURE TAsyncOp.Fields(PROCEDURE DoToField(
16:         fieldName: Str255;
17:         fieldAddr: Ptr;
18:         fieldType: INTEGER)); OVERRIDE;
19:   END; {TAsyncOp}
```

Note the state variable at line 4. This can take on two values as defined on line 1: idling and waiting. The **TAsyncOp** is idling when no polling is being performed and waiting when polling is being performed.

Also note the **fFreqWhenWaiting** field on line 5. This is set to a default value at initialization but can be reset by calling the method **UseIdleFreq**. This is the frequency at which the polling will take place.

TAsyncOp.ITAsyncOp

The **ITAsyncOp** method is used to initialize the object. Listing 12-2 shows this method.

Listing 12-2. The ITAsyncOP method (TAsyncOp.ITAsyncOp)

```
{$S AInit}
PROCEDURE TAsyncOp.ITAsyncOp;
BEGIN {TAsyncOp.ITAsyncOp}
    fState := kIdling;
    fIdleFreq := kMaxIdleTime;
    fFreqWhenWaiting := 1;
    gApplication.InstallCohandler(SELF,TRUE);
END; {TAsyncOp.ITAsyncOp}
```

ITAsyncOp begins by initializing the three instance variables **fState**, **fIdleFreq**, and **fFreqWhenWaiting** on lines 4–6. Note that setting the **fIdleFreq** to **kMaxIdleTime** really means that this object will not get idle time until the idle frequency.is changed.

Line 7 installs this object into the cohandler chain.

TAsyncOp.DoOperation

The **DoOperation** method is called to indicate that an asynchronous operation has been started and that polling should begin. In order for this object's idle method to be called, the **fIdleFreq** must be changed from **kMaxIdleFreq** to a smaller number.

Listing 12-3 illustrates this method.

Listing 12-3. The DoOperation method (TAsyncOp.DoOperation)

```
1: {$S ARes}
2: PROCEDURE TAsyncOp.DoOperation;
3: BEGIN {TAsyncOp.DoOperation}
4:     fState    := kWaiting;
5:     fIdleFreq := fFreqWhenWaiting;
6: END; {TAsyncOp.DoOperation}
```

Line 4 changes the state of the object to waiting to indicate that polling is being done. Line 5 sets it to the desired frequency.

TAsyncOp.PollForCompletion

The **PollForCompletion** method (shown in Listing 12-4) is really a placeholder for a method that must be supplied by the subclass. This version of it simply always returns FALSE. The subclass must override this method and implement the proper polling behavior for itself. This method typically implements a check for the **ioResult** field of a parameter block.

Listing 12-4. The PollForCompletion method (TAsyncOp.PollForCompletion)

```
1: {$S ARes}
2: FUNCTION TAsyncOp.PollForCompletion : BOOLEAN;
3: BEGIN {TAsyncOp.PollForCompletion}
4:     PollForCompletion := FALSE;
5: END; {TAsyncOp.PollForCompletion}
```

TAsyncOp.DoAfterCompletion

The **DoAfterCompletion** method is also a placeholder for a method that must be supplied by the subclass. But unlike **PollForCompletion,** this method must be called by the subclass' method so that the instance variables can be reset to the proper values. Listing 12-5 illustrates the **DoAfterCompletion** method.

Listing 12-5. The DoAfterCompletion method (TAsyncOp.DoAfterCompletion)

```
{$S ARes}
PROCEDURE TAsyncOp.DoAfterCompletion;
BEGIN {TAsyncOp.DoAfterCompletion}
    fState    := kIdling;
    fIdleFreq := kMaxIdleTime;
END; {TAsyncOp.DoAfterCompletion}
```

TAsyncOp.UseIdleFreq

The **UseIdleFreq** method is used to reset the idle frequency to some value other than the default. This method is rarely overridden. Listing 12-6 displays the **UseIdleFreq** method.

Listing 12-6. TAsyncOp.UseIdleFreq

```
1: {$S ARes}
2: PROCEDURE TAsyncOp.UseIdleFreq(newFreq : LONGINT);
3: BEGIN {TAsyncOp.UseIdleFreq}
4:     fFreqWhenWaiting := newFreq;
5:     IF fState = kWaiting
6:         THEN BEGIN
7:             fIdleFreq := newFreq;
8:         END;
9: END; {TAsyncOp.UseIdleFreq}
```

Line 4 sets the waiting frequency to the new value, and line 7 sets the current idle frequency to the new value the object is currently polling.

TAsyncOp.Abort

The **Abort** method is meant to be overridden to provide a way to abort the current asynchronous operation. Listing 12-7 illustrates the **Abort** method.

Listing 12-7. The Abort method (TAsyncOp.Abort)

```
{$S ARes}
PROCEDURE TAsyncOp.Abort;
BEGIN {TAsyncOp.Abort}

END; {TAsyncOp.Abort}
```

TAsyncOp.Free

The **Free** method is used to abort any pending operations and to remove the object from the cohandler list. Because no dynamic memory allocation is used, none is freed here. If a subclass of **TAsyncOp** used dynamically allocated storage, it should be reclaimed in this method. Listing 12-8 shows the **Free** method.

Listing 12-8. The Free method (TAsyncOp.Free)

```
{$S AFree}
PROCEDURE TAsyncOp.Free; OVERRIDE;
BEGIN {TAsyncOp.Free}
    IF fState = kWaiting
        THEN BEGIN
            SELF.Abort;
        END;
    gApplication.InstallCohandler(SELF,FALSE);
    SELF.Free;
END; {TAsyncOp.Free}
```

TAsyncOp.DoIdle

The **DoIdle** method is the meat of the polling process. It is called when the application has idle time and at least **fIdleFreq** time has passed since the last time the idle method was called. Listing 12-9 illustrates the **TAsyncOp.DoIdle** method.

Listing 12-9. TAsyncOp.DoIdle

```
 1: {$S ARes}
 2: FUNCTION TAsyncOp.DoIdle(phase: IdlePhase) :
 3:                 BOOLEAN; OVERRIDE;
 4: BEGIN {TAsyncOp.DoIdle}
 5:     IF phase = IdleContinue { others: idleBegin, idleEnd }
 6:         THEN BEGIN
 7:             IF fState = kWaiting
 8:                 THEN BEGIN
 9:                     REPEAT
10:                         IF SELF.PollForCompletion
11:                             THEN BEGIN
12:                                 DoAfterCompletion;
13:                             END
14:                             ELSE BEGIN
15:                                 Leave;
16:                             END;
17:                     UNTIL fState <> kWaiting;
18:                 END;
19:         END;
20: END; {TAsyncOp.DoIdle}
```

Line 5 checks for the **IdleContinue** idle phase. This indicates that real idle time is available and not something else (like a menu being pressed).

Line 7 checks that the object is in fact in the waiting state.

Line 10 checks to see if the object is done polling. If it is, line 12 calls the **DoAfterCompletion** method to perform post-processing.

The extra check of **PollForCompletion** is placed in the loop from lines 9–17 to handle the case where the **DoAfterCompletion** method chains another operation right away. It results in another polling check without having to wait for the idle method to be called again. With this scheme, a series of operations can be performed quickly with no wasted time in-between.

▶ The TADSP Abstract Class

The TADSP abstract class implements a basic ADSP class on top of the **TAsyncOp** class. Its methods implement common ADSP functionality that is used by all of its subclasses.

TADSP allocates and deallocates all the various buffers needed by ADSP. TADSP opens and initializes a connection end as well as implements the **PollForCompletion** method for ADSP operations. TADSP does this by checking the **ioResult** field of the ADSP parameter block.

Another feature of the TADSP class is that it can register and unregister an NBP name on the socket used by the connection end. Listing 12-10 shows the declaration for the TADSP class.

Listing 12-10. Declaration of the class TADSP

```
TADSP = OBJECT(TAsyncOp)
  fADSPSocket          : integer;
  fADSP                : DSPPBPtr;
  fCCBPtr              : TPCCB;
  fSendQueue           : Ptr;
  fRecvQueue           : Ptr;
  fAttnPtr             : Ptr;
  fADSPData            : Ptr;
  fNTE                 : ^NamesTableEntry;
  fNBPName             : str32;
  fNBPType             : str32;
  fNameIsRegistered    : boolean;
  PROCEDURE TADSP.ITADSP(NBPName : str255;
                         NBPType : str255);
  FUNCTION  TADSP.PollForCompletion : BOOLEAN;
      OVERRIDE;
  PROCEDURE TADSP.Free; OVERRIDE;
  PROCEDURE TADSP.RegisterName;
  PROCEDURE TADSP.UnRegisterName;
  PROCEDURE TADSP.Fields(PROCEDURE DoToField(
                  fieldName: Str255;
                  fieldAddr: Ptr;
                  fieldType: INTEGER)); OVERRIDE;
END; {TADSP}
```

TADSP.ITADSP

The **ITADSP** method is used to initialize the object. Listing 12-11 shows this method. It begins by calling the initialization method for its parent class—**TAsyncOP**—on line 18.

Listing 12-11. The ITADSP method (TADSP.ITADSP)

```
{$S AInit}
PROCEDURE TADSP.ITADSP(NBPName : str255; NBPType :
str255);
VAR
    myMPPPB : MPPParamBlock;
    s       : str255;
PROCEDURE InitADSP;
VAR
    dummy   : integer;
BEGIN {InitNet}
    FailOSErr(OpenDriver('.MPP',dummy));
    IF OpenDriver('.DSP',gADSP) <> noErr
        THEN BEGIN
            StdAlert(kADSPAbsent);
            ExitMacApp;
        END;
END; {InitNet}
BEGIN {TADSP.ITADSP}
    SELF.ITAsyncOp;
    InitADSP;
    fNameIsRegistered := FALSE;
    IF NBPName = ''
        THEN BEGIN
            NBPName := GetString(kChooserNameStrID)^^;
        END;
    fNBPName := NBPName;
    fNBPType := NBPType;
    fADSP := DSPPBPtr(NewPtr(SIZEOF(DSPParamBlock)));
    FailNil(fADSP);
    fADSPData := NewPtr(kADSPMaxCommand);
    FailNil(fADSPData);
    WITH fADSP^ DO
        BEGIN
            ccbPtr := TPCCB(NewPtr(SIZEOF(TRCCB)));
            FailNil(ccbPtr);
            sendQueue := NewPtr(kADSPSendBufSize);
            FailNil(sendQueue);
            recvQueue := NewPtr(kADSPRecvBufSize);
            FailNil(recvQueue);
            attnPtr := NewPtr(attnBufSize);
            FailNil(attnPtr);
            ioCompletion:= NIL;
            userRoutine := NIL;
            ioCRefNum   := gADSP;
```

```
44:             fCcbPtr     := ccbPtr;
45:             sendQSize   := kADSPSendBufSize;
46:             recvQSize   := kADSPRecvBufSize;
47:             fSendQueue  := sendQueue;
48:             fRecvQueue  := recvQueue;
49:             fAttnPtr    := attnPtr;
50:             localSocket := 0;
51:             csCode      := dspInit;
52:         END;
53:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
54:     fADSPSocket := fADSP^.localSocket;
55: END; {TADSP.ITADSP}
```

Line 19 calls the routine **InitADSP**. The routine is shown on lines 6–16 and it opens the .MPP and .ADSP drivers.

Line 20 sets the **fNameIsRegistered** to an initial value of FALSE because it isn't registered yet.

Line 21 checks for a supplied NBP name. If none was supplied, it gets the user-name string from the System file. Line 26 sets the NBP type to the string supplied.

Line 27 allocates the ADSP parameter block and line 28 allocates a general purpose buffer for use when sending or receiving data.

Lines 33–40 allocate the send, receive, and attention buffers as well as the connection control block.

Lines 41–49 fill in the ADSP parameter block with appropriate values.

Line 50 indicates that any free socket should be allocated to this connection end by ADSP.

Line 51 tells the .ADSP driver that it should initialize a connection end.

Line 53 makes the call to the driver.

Line 54 saves the socket number returned by the **dspInit** call into the **fSocket** instance variable for use later.

TADSP.PollForCompletion

The **PollForCompletion** method implements the proper polling action for asynchronous ADSP operations. It checks the **ioResult** field of the ADSP parameter block for any value other than 1. Listing 12-12 shows this method.

Listing 12-12. The PollForCompletion method (TADSP.PollForCompletion)

```
{$S ARes}
FUNCTION TADSP.PollForCompletion : boolean; OVERRIDE;
BEGIN {TADSP.PollForCompletion}
    PollForCompletion := (fADSP^.ioResult <> 1);
END; {TADSP.PollForCompletion}
```

TADSP.Free

The **Free** method deallocates all dynamic storage used by the TADSP object. It also closes down the connection end and unregisters the NBP name. Listing 12-13 shows the **Free** method.

Listing 12-13. The Free method (TADSP.Free)

```
 1: {$S AFree}
 2: PROCEDURE TADSP.Free; OVERRIDE;
 3: VAR
 4:     stat        : OSErr;
 5:     theDSPPB    : DSPParamBlock;
 6: BEGIN {TADSP.Free}
 7:     UnRegisterName;
 8:     theDSPPB := fADSP^;
 9:     WITH theDSPPB DO BEGIN
10:         abort  := 1;
11:         csCode := dspRemove;
12:     END;
13:     stat := PBControl(@theDSPPB,kSYNC);
14:     DisposPtr(Ptr(fCcbPtr));
15:     DisposPtr(fSendQueue);
16:     DisposPtr(fRecvQueue);
17:     DisposPtr(fAttnPtr);
18:     DisposPtr(Ptr(fADSP));
19:     DisposPtr(fADSPData);
20:     INHERITED Free;
21: END; {TADSP.Free}
```

Line 7 unconditionally calls the **UnRegisterName** method. No check for a name being registered is done here because the **UnRegisterName** method does that itself.

Next, lines 8–13 tell the .ADSP driver to close the connection end. Lines 14–19 deallocate all dynamic storage used by the object. Line 20 calls the superclass' **Free** method so it can deallocate its storage.

TADSP.RegisterName

The **RegisterName** method registers a name for the connection end's socket. Listing 12-14 illustrates the **RegisterName** method.

Listing 12-14. The TADSP.RegisterName

```
 1: {$S ARes}
 2: PROCEDURE TADSP.RegisterName;
 3: VAR
 4:     tNameStr    : str255;
 5:     tTypeStr    : str255;
 6:     theMPPPB    : MPPParamBlock;
 7: BEGIN {TADSP.RegisterName}
 8:     IF fNameIsRegistered
 9:         THEN BEGIN
10:             UnRegisterName;
11:         END;
12:     fNTE := Pointer(NewPtr(SIZEOF(NamesTableEntry)));
13:     FailNil(fNTE);
14:     tNameStr := fNBPName;
15:     tTypeStr := fNBPType;
16:     NBPSetNTE(Ptr(fNTE),tNameStr,tTypeStr,'*',
        fADSPSocket);
17:     WITH theMPPPB DO BEGIN
18:         interval    := kNBPTimeOutVal;
19:         count       := kNBPRetryCount;
20:         entityPtr   := Ptr(fNTE);
21:         verifyFlag  := 1;
22:     END;
23:     FailOSErr(PRegisterName(@theMPPPB,kSYNC));
24:     fNameIsRegistered := TRUE;
25: END; {TADSP.RegisterName}
```

TADSP.RegisterName first checks for a previously registered name and unregisters it if there is one on lines 8–11.

Line 12 allocates a names table entry for the name and lines 14–16 create it.

Lines 17–22 set up a parameter block for registering the name.
Line 23 calls the **PRegisterName** routine to actually register the name.
If line 24 is reached, the name has been registered so the flag **fNameIsRegistered** is set to TRUE.

TADSP.UnRegisterName

The **UnRegisterName** method unregisters a name if there is an NBP name registered.

It first checks to see if there is a registered name. If there is, it calls **PRemoveName** for that names table entry. After the name is removed, it sets the **fNameIsRegistered** flag to FALSE. Listing 12-15 displays the **UnRegisterName** method.

Listing 12-15. The UnRegisterName method (TADSP.UnRegisterName)

```
{$S ARes}
PROCEDURE TADSP.UnRegisterName;
VAR
    theMPPPB    : MPPParamBlock;
    stat        : OSErr;
BEGIN {TADSP.UnRegisterName}
    IF fNameIsRegistered
        THEN BEGIN
            theMPPPB.entityPtr :=
                        Ptr(ord4(@fNTE^.nteData)+1);
            stat := PRemoveName(@theMPPPB,kSYNC);
            fNameIsRegistered := FALSE;
        END;
END; {TADSP.UnRegisterName}
```

▶ The TPlayer Abstract Class

The **TPlayer** abstract class implements the player message handling. It has methods for sending the outgoing player messages as well as handling the incoming message. It also keeps track of the player state with the **fPlayerState** instance variable.

TPlayer provides the method **DoAfterConnection** for its subclass to override so that it can perform the proper action at that point.

Listing 12-16 shows the declaration for the **TPlayer** class.

Listing 12-16. Declaration of the TPlayer class

```
PlayerStates = (kConnecting,kMyTurn,kYourTurn);

CheckersMessages = (kStartTracking,kTrack,
  kStopTracking,kUpdateBoard,kYieldTurn);

TPlayer = OBJECT(TADSP)
  fPlayerState : PlayerStates;
  PROCEDURE TPlayer.ITPlayer;
  PROCEDURE TPlayer.StartTracking(
      whichPiece : GamePieceSpecifier);
  PROCEDURE TPlayer.Track(oldRect : FakeVRectPtr
                          newRect : FakeVRectPtr);
  PROCEDURE TPlayer.StopTracking;
  PROCEDURE TPlayer.YieldTurn(theGameState : GameState);
  PROCEDURE TPlayer.SendBoardState;
  PROCEDURE TPlayer.ReadAMessage;
  PROCEDURE TPlayer.HandleIncomingMessage;
  PROCEDURE TPlayer.DoAfterCompletion; OVERRIDE;
  PROCEDURE TPlayer.DoAfterConnection;
  PROCEDURE TPlayer.Fields(PROCEDURE DoToField(
              fieldName: Str255;
              fieldAddr: Ptr;
              fieldType: INTEGER)); OVERRIDE;
END; {TPlayer}
```

TPlayer.ITPlayer

The **ITPlayer** method is used to initialize the object.

All **ITPlayer** does is call its initialization routine for the parent class with the checkers application NBP type and no name, forcing the Chooser name to be used. Listing 12-17 shows the **ITPlayer** method.

Listing 12-17. The ITPlayer method (TPlayer.ITPlayer)

```
{$S AInit}
PROCEDURE TPlayer.ITPlayer;
BEGIN {TPlayer.ITPlayer}
    ITADSP('',KCheckersNBPType);
END; {TPlayer.ITPlayer}
```

TPlayer.StartTracking

The **StartTracking** method is called when a player starts moving a checker by clicking and dragging. It signals to the other end that it should get ready to accept track messages. Listing 12-18 displays the **StartTracking** method.

Listing 12-18. The StartTracking method (TPlayer.StartTracking)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.StartTracking(
 3:                 whichPiece : GamePieceSpecifier);
 4: VAR
 5:     theMessage : CheckersMessages;
 6: BEGIN {StartTracking}
 7:     theMessage := kStartTracking;
 8:     WITH fADSP^ DO BEGIN
 9:         dataPtr  := @theMessage;
10:         reqCount := SIZEOF(CheckersMessages);
11:         eom      := 0;
12:         flush    := 0;
13:         csCode   := dspWrite;
14:     END;
15:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
16:     WITH fADSP^ DO BEGIN
17:         dataPtr  := @whichPiece;
18:         reqCount := SIZEOF(GamePieceSpecifier);
19:         eom      := 1;
20:         flush    := 1;
21:         csCode   := dspWrite;
22:     END;
23:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
24: END; {StartTracking}
```

The message is sent in two parts. First, the kind of message being sent is written to the ADSP connection. This is done on lines 7–15.

Then the data that goes along with it is sent, in this case specifying which game piece is being moved. This is done on lines 16–23.

Note that the first part of the message is sent with the **eom** and **flush** flags set to zero. This allows ADSP to buffer it and not necessarily send it right away. When the last part of the message is sent, the **eom** and **flush** flags are set to 1; this tells ADSP to send it right away.

The **eom** flag doesn't really need to be set, but is helpful when debugging. If the size of the sent messages and the expected sizes don't match, setting the **eom** flag can help synchronize things.

TPlayer.Track

The **Track** method is called as the player is moving the checker by dragging. It allows the other end to track the checker in real-time. Listing 12-19 displays the **Track** method.

Listing 12-19. The TPlayer.Track method (TPlayer.Track)

```
{$S ARes}
PROCEDURE TPlayer.Track(oldRect,newRect : FakeVRectPtr);
VAR
    theMessage : CheckersMessages;
BEGIN {Track}
    theMessage := kTrack;
    WITH fADSP^ DO BEGIN
        dataPtr  := @theMessage;
        reqCount := SIZEOF(CheckersMessages);
        eom      := 0;
        flush    := 0;
        csCode   := dspWrite;
    END;
    FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
    WITH fADSP^ DO BEGIN
        dataPtr  := @oldRect;
        reqCount := SIZEOF(FakeVRectPtr);
        eom      := 0;
        flush    := 0;
        csCode   := dspWrite;
    END;
    FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
    WITH fADSP^ DO BEGIN
        dataPtr  := @newRect;
        reqCount := SIZEOF(FakeVRectPtr);
        eom      := 1;
        flush    := 1;
        csCode   := dspWrite;
    END;
    FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
END; {Track}
```

The message is sent in three parts. First the kind of message being sent is written to the ADSP connection. This is done on lines 6–14.

Then the data that describes the old checker location and the new checker location is sent. This is done by writing each **VRect** to the connection on lines 15–30.

Note that the first part of the message is sent with the **eom** and **flush** flags set to zero. This allows them to buffer up and not send right away. When the last part of the message is sent, the **eom** and **flush** flags are set to 1; this tells ADSP to send it right away.

TPlayer.StopTracking

The **StopTracking** method is called when the player stops dragging the checker and has released the mouse button. It allows the other end to know that the checker tracking operation is complete. Listing 12-20 shows the **StopTracking** method.

Listing 12-20. The StopTracking method (TPlayer.StopTracking)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.StopTracking;
 3: VAR
 4:     theMessage : CheckersMessages;
 5: BEGIN {StopTracking}
 6:     theMessage := kStopTracking;
 7:     WITH fADSP^ DO BEGIN
 8:         dataPtr  := @theMessage;
 9:         reqCount := SIZEOF(CheckersMessages);
10:         eom      := 1;
11:         flush    := 1;
12:         csCode   := dspWrite;
13:     END;
14:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
15: END; {StopTracking}
```

The message is short because it only indicates the message kind and no other data is sent. The message kind is written to the connection in lines 6–14. The **eom** and **flush** flags are set to 1 forcing the end of message so that the message is sent right away.

TPlayer.YieldTurn

The **YieldTurn** method is called when a player's turn is over. It signals to the other end that it should take its turn. Listing 12-21 illustrates the **YieldTurn** method.

Listing 12-21. The YieldTurn method (TPlayer.YieldTurn)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.YieldTurn(theGameState : GameState);
 3: VAR
 4:     theMessage : CheckersMessages;
 5: BEGIN {YieldTurn}
 6:     fPlayerState := kYourTurn;
 7:     theMessage := kYieldTurn;
 8:     WITH fADSP^ DO BEGIN
 9:         dataPtr  := @theMessage;
10:         reqCount := SIZEOF(CheckersMessages);
11:         eom      := 0;
12:         flush    := 0;
13:         csCode   := dspWrite;
14:     END;
15:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
16:     WITH fADSP^ DO BEGIN
17:         dataPtr  := @theGameState;
18:         reqCount := SIZEOF(GameState);
19:         eom      := 1;
20:         flush    := 1;
21:         csCode   := dspWrite;
22:     END;
23:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
24: END; {YieldTurn}
```

The message is sent in two parts. First, the kind of message being sent is written to the ADSP connection. This is done on lines 5–15. Note how on line 6 the player state gets set to **kYourTurn** indicating that this player has yielded its turn.

Then the part of the message indicating whether the current player has determined if the game is over is sent. This is done on lines 16–23.

TPlayer.SendBoardState

The **SendBoardState** method is called before the turn is yielded. It tells the remote player about the new board setup. The entire board setup is sent to ensure that the two sides don't accidentally end up with different representations of the game board. Listing 12-22 illustrates the **SendBoardState** method.

Listing 12-22. The SendBoardState method (TPlayer.SendBoardState)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.SendBoardState;
 3: VAR
 4:     theMessage : CheckersMessages;
 5: BEGIN {TPlayer.SendBoardState}
 6:     theMessage := kUpdateBoard;
 7:     WITH fADSP^ DO BEGIN
 8:         dataPtr  := @theMessage;
 9:         reqCount := SIZEOF(CheckersMessages);
10:         eom      := 0;
11:         flush    := 0;
12:         csCode   := dspWrite;
13:     END;
14:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
15:     WITH fADSP^ DO BEGIN
16:         dataPtr  := @gBoardData;
17:         reqCount := SIZEOF(BoardData);
18:         eom      := 1;
19:         flush    := 1;
20:         csCode   := dspWrite;
21:     END;
22:     FailOSErr(PBControl(ParmBlkPtr(fADSP),kSYNC));
23: END; {TPlayer.SendBoardState}
```

The message is sent in two parts. First the kind of message being sent is written to the ADSP connection. This is done on lines 6–14.

Then the part of the message containing the board state is sent. This is done on lines 15–22.

TPlayer.ReadAMessage

The **ReadAMessage** method is called to start an asynchronous read on the connection. Listing 12-23 displays the **ReadAMessage** method.

Listing 12-23. The ReadAMessage method (TPlayer.ReadAMessage)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.ReadAMessage;
 3: VAR
 4:     stat    : OSErr;
 5: BEGIN {TPlayer.ReadAMessage}
 6:     WITH fADSP^ DO BEGIN
 7:         dataPtr  := @fADSPData;
 8:         reqCount := SIZEOF(CheckersMessages);
 9:         csCode   := dspRead;
10:     END;
11:     stat := PBControl(ParmBlkPtr(fADSP),kASYNC);
12:     IF SELF.PollForCompletion
13:         THEN BEGIN
14:             DoAfterCompletion;
15:         END
16:         ELSE BEGIN
17:             DoOperation;
18:         END;
19: END; {TPlayer.ReadAMessage}
```

ReadAMessage hangs the read on the connection on lines 6–11. Then, rather than wait for the next idle time to check for completion, it calls **PollForCompletion** itself right away. It this returns TRUE, indicating the read has already completed, it calls **DoOperation** on line 17 to process the read. This leads to a much more efficient read process than waiting for idle time each time a read is issued.

TPlayer.DoAfterCompletion

The **DoAfterCompletion** method is used to process the read operation after it has completed. Listing 12-24 shows the **DoAfterCompletion** method.

Listing 12-24. The DoAfterCompletion method (TPlayer.DoAfterCompletion)

```
 1: {$S ARes}
 2: PROCEDURE TPlayer.DoAfterCompletion; OVERRIDE;
 3: BEGIN {TPlayer.DoAfterCompletion}
 4:     INHERITED DoAfterCompletion;
 5:     CASE fPlayerState OF
 6:         kConnecting : BEGIN
 7:             IF fADSP^.ioResult = noErr
 8:                 THEN BEGIN
 9:                     DoAfterConnection;
10:                 END
11:                 ELSE BEGIN
12:                     DebugStr('open connection error');
13:                 END;
14:         END;
15:         otherwise   BEGIN
16:             IF fADSP^.ioResult = noErr
17:                 THEN BEGIN
18:                     IF (fADSP^.actCount = 0) AND
19:                         (fCcbPtr^.state = sClosed)
20:                         THEN BEGIN
21:                             SysBeep(5);
22:                         END
23:                         ELSE BEGIN
24:                             HandleIncomingMessage;
25:                             ReadAMessage;
26:                         END;
27:                 END
28:                 ELSE BEGIN
29:                     DebugStr('adsp read error');
30:                 END;
31:         END;
32:     END;
33: END; {TPlayer.DoAfterCompletion}
```

DoAfterCompletion handles two cases. The first is the completion of a connection request. This is indicated by the **fPlayerState** being set to **kConnecting** and is handled by calling the **DoAfterConnection** method at line 9 if no error is encountered.

The second case is if a read has been issued on the connection. This is indicated by the **fPlayerState** not being set to **kConnecting**. This is handled on lines 15–31.

If the connection has not been closed, checked on lines 18 and 19, and only indicated by a beep at line 21, the **HandleIncomingMessage** method is called to process the message and the **ReadAMessage** method is called to reissue the read request.

TPlayer.DoAfterConnection

The **DoAfterConnection** method is a placeholder for the subclasses. The subclass will implement this method to perform the actions that are needed when an open connection is completed. It must be overridden to do anything useful. Listing 12-25 displays the **DoAfterConnection** method.

Listing 12-25. The DoAfterConnection method (TPlayer.DoAfterConnection)

```
{$S ARes}
PROCEDURE TPlayer.DoAfterConnection;
BEGIN {TPlayer.DoAfterConnection}

END; {DoAfterConnection}
```

TPlayer.HandleIncomingMessage

The **HandleIncomingMessage** method processes all the incoming messages. Each type of message is processed separately. Note that in order to provide real-time tracking of the checkers once a **StartTracking** message is received, this code performs its own reads of the Track messages until a **StopTracking** message is received. Listing 12-26 illustrates the **HandleIncomingMessage** routine.

Listing 12-26. The HandleIncoming Message method (TPlayer.HandleIncomingMessage)

```
{$S ARes}
PROCEDURE TPlayer.HandleIncomingMessage;
VAR
    theGameState: GameState;
    first       : boolean;
    stat        : OSErr;
    inRect      : FakeVRect;
    oldRect     : FakeVRect;
    incomingMessageKind : CheckersMessages;
    whichGamePiece      : GamePieceSpecifier;
BEGIN {TPlayer.HandleIncomingMessage}
  BlockMove(fADSPData,@incomingMessageKind,
                            SIZEOF(CheckersMessages));
  CASE incomingMessageKind OF
    kStartTracking : BEGIN
        WITH fADSP^ DO BEGIN
            dataPtr  := fADSPData;
            reqCount := SIZEOF(GamePieceSpecifier);
            csCode   := dspRead;
        END;
        stat := PBControl(ParmBlkPtr(fADSP),kSYNC);
        BlockMove(fADSPData,@whichGamePiece,
                        SIZEOF(GamePieceSpecifier));
        StartMoving(whichGamePiece);
        first := TRUE;
        HideCursor;
        REPEAT
            WITH fADSP^ DO BEGIN
                dataPtr  := fADSPData;
                reqCount := SIZEOF(CheckersMessages);
                csCode   := dspRead;
            END;
            stat := PBControl(ParmBlkPtr(fADSP),kSYNC);
            BlockMove(fADSPData,@incomingMessageKind,
                            SIZEOF(CheckersMessages));
            IF incomingMessageKind = kTrack
                THEN BEGIN
                    WITH fADSP^ DO BEGIN
                        dataPtr  := fADSPData;
                        reqCount := SIZEOF(FakeVRect);
                        csCode   := dspRead;
                    END;
                    stat :=
                       PBControl(ParmBlkPtr(fADSP),kSYNC);
                    BlockMove(fADSPData,@inRect,
                                    SIZEOF(FakeVRect));
                    WITH fADSP^ DO BEGIN
                        dataPtr  := fADSPData;
                        reqCount := SIZEOF(FakeVRect);
                        csCode   := dspRead;
                    END;
                    stat :=
                       PBControl(ParmBlkPtr(fADSP),kSYNC);
                    BlockMove(fADSPData,@oldRect,
                                    SIZEOF(FakeVRect));
                    IF first
                        THEN BEGIN
                            LocateIt(inRect,FALSE,FALSE);
                            first := FALSE;
                        END;
                    MoveIt(inRect,oldRect);
                    YieldTime;
                END;
        UNTIL incomingMessageKind <> kTrack;
        LocateIt(inRect,FALSE,TRUE);
        ShowCursor;
    END;
    kTrack : BEGIN
        DebugStr('Rogue kTrack Message');
    END;
    kStopTracking : BEGIN
        DebugStr('Rogue kStopTracking Message');
    END;
    kUpdateBoard : BEGIN
        WITH fADSP^ DO BEGIN
            dataPtr  := fADSPData;
            reqCount := SIZEOF(BoardData);
            csCode   := dspRead;
        END;
        stat := PBControl(ParmBlkPtr(fADSP),kSYNC);
        BlockMove(fADSPData,@gBoardData,
                            SIZEOF(BoardData));
        UpdateBoard(gBoardData);
    END;
    kYieldTurn : BEGIN
        WITH fADSP^ DO BEGIN
            dataPtr  := fADSPData;
            reqCount := SIZEOF(GameState);
            csCode   := dspRead;
        END;
        stat := PBControl(ParmBlkPtr(fADSP),kSYNC);
        BlockMove(fADSPData,@theGameState,SIZEOF
        (boolean));
        MyTurn(theGameState);
        fPlayerState := kMyTurn;
    END;
  END;
END; {TPlayer.HandleIncomingMessage}
```

Line 12 copies the data read from the objects buffer into the **incomingMessageKind** variable. This is then used on line 14 to determine what kind of message has been received and proper processing is done.

Lines 15–67 process the **StartTracking** message. It first reads in which game piece is specified on lines 16–23. Then it calls the **StartMoving** procedure to tell the application that the specified piece will be moving.

Next it goes into a loop to read Track messages. This loop is exited at line 64 when a message that isn't a Track message is received.

Lines 28–35 read in the message kind. If it's a Track message, then lines 38–63 handle it. Otherwise the loop exits.

If a Track message is received, the two **VRects** describing the before and after position are read in on lines 38–55. If it's the first time through, the **LocateIt** procedure is called to fix the checkers position.

Then the **MoveIt** procedure is called on line 61 to actually move the checker.

Yield time is called on line 62 to give the background tasks some time.

Once the loop is exited, the **LocateIt** call is called again to fix the final location of the checker.

If either the **Track** or **StopTracking** messages are received outside of the loop in **StartTracking**, this is because of an error. So lines 68–73 call the debugger to indicate this.

Lines 78–84 handle the **UpdateBoard** message, which reads in the board data and calls the **UpdateBoard** procedure to have the application set the board to the state received from the other side.

Lines 85–95 handle the **YieldTurn** message, which reads in the game state data and calls the application's **MyTurn** procedure to indicate that it is this player's turn now.

▶ The TActivePlayer and TPassivePlayer Classes

The names **TActivePlayer** and **TPassivePlayer** only refer to the mode they use to open the connection. Both players actively play the game. The **TPlayer** class is used as the superclass for the two concrete classes that are used to implement the checkers game. Each of these classes is very similar because they inherit most of their behavior from the **TPlayer** class.

TActivePlayer and **TPassivePlayer** implement two things in addition to the operations found in **TPlayer**. First, they both open the connection slightly differently, one in request mode, the other in passive mode. The second difference is that **TActivePlayer** always takes its turn first, while **TPassivePlayer** always goes second.

Listings 12–27 and 12–28 list the declarations of the **TActivePlayer** and **TPassivePlayer** classes. The only real difference between the two classes is that they call their connection methods by different names.

Listing 12-27. Declaration of the TActivePlayer class

```
TActivePlayer = OBJECT(TPlayer)
  PROCEDURE TActivePlayer.ITActivePlayer;
  PROCEDURE TActivePlayer.OpenConnection;
  PROCEDURE TActivePlayer.DoAfterConnection; OVERRIDE;
  PROCEDURE TActivePlayer.Fields(PROCEDURE DoToField(
              fieldName: Str255;
              fieldAddr: Ptr;
              fieldType: INTEGER)); OVERRIDE;
END; {TActivePlayer}
```

Listing 12-28. Declaration of the TPassivePlayer class

```
TPassivePlayer = OBJECT(TPlayer)
  PROCEDURE TPassivePlayer.ITPassivePlayer;
  PROCEDURE TPassivePlayer.OpenPassiveConnection;
  PROCEDURE TPassivePlayer.DoAfterConnection; OVERRIDE;
  PROCEDURE TPassivePlayer.Fields(PROCEDURE DoToField(
              fieldName: Str255;
              fieldAddr: Ptr;
              fieldType: INTEGER)); OVERRIDE;
END; {TPassivePlayer}
```

TActivePlayer.ITActivePlayer

The **ITActivePlayer** method is used to initialize the object. It is called right after the object is created.

ITActivePlayer calls its parent class initialization method, then the **OpenConnection** method to get the connection going for this object. Listing 12-29 illustrates the **ITActivePlayer** method.

Listing 12-29. The ITActivePlayer method (TActivePlayer.ITActivePlayer)

```
{$S AInit}
PROCEDURE TActivePlayer.ITActivePlayer;
BEGIN {TActivePlayer.ITActivePlayer}
    ITPlayer;
    OpenConnection;
END; {TActivePlayer.ITActivePlayer}
```

TPassivePlayer.ITPassivePlayer

The **ITPassivePlayer** method is used to initialize the object. It is called right after the object is created.

ITPassivePlayer calls its parent class initialization method, then the **OpenPassiveConnection** method to get the connection going for this object. It also registers its name on the network so that the other player can find it. Listing 12-30 illustrates the **ITPassivePlayer** method.

Listing 12-30. The ITPassivePlayer method (TPassivePlayer.ITPassivePlayer)

```
{$S AInit}
PROCEDURE TPassivePlayer.ITPassivePlayer;
BEGIN {TPassivePlayer.ITPassivePlayer}
    ITPlayer;
    OpenPassiveConnection;
    RegisterName;
END; {TPassivePlayer.ITPassivePlayer}
```

TActivePlayer.OpenConnection

The **OpenConnection** method opens the connection for the active player. This method is displayed in Listing 12-31.

Listing 12-31. The OpenConnection method (TActivePlayer.OpenConnection)

```
 1: {$S ARes}
 2: PROCEDURE TActivePlayer.OpenConnection;
 3: VAR
 4:     stat             : OSErr;
 5:     theName,theZone : str255;
 6:     theAddr          : addrBlock;
 7: BEGIN {TActivePlayer.OpenConnection}
 8:     IF SelectName('=','=',theName,theZone, theAddr)
 9:         THEN BEGIN
10:             WITH fADSP^ DO BEGIN
11:                 remoteAddress := theAddr;
12:                 filterAddress := AddrBlock(0);
13:                 ocMode        := ocRequest;
14:                 ocInterval    := 0;
15:                 ocMaximum     := 0;
16:                 csCode        := dspOpen;
17:             END;
18:             stat := PBControl(ParmBlkPtr(fADSP),kASYNC);
19:             fPlayerState := kConnecting;
20:             DoOperation;
21:         END;
22: END; {TActivePlayer.OpenConnection}
```

OpenConnection starts by calling the **SelectName** function on line 8. This routine puts up a Chooser-like dialog that lets the user connect to another checkers player.

If the user selects a remote player, then lines 10–18 issue an asynchronous **dspOpen** to the remote player using the request mode.

Line 19 sets the player state to connecting and line 20 calls **DoOperation** to start the polling. This uses the **TAsyncOp** technique for handling asynchronous operations during idle times.

TPassivePlayer.OpenPassiveConnection

The **OpenPassiveConnection** method opens the connection for the passive player. It issues an asynchronous **dspOpen** call using the passive mode. Listing 12-32 shows the **OpenPassiveConnection** method.

Listing 12-32. The OpenPassiveConnection method (TPassivePlayer.OpenPassiveConnection)

```
 1: {$S ARes}
 2: PROCEDURE TPassivePlayer.OpenPassiveConnection;
 3: VAR
 4:     stat    : OSErr;
 5: BEGIN {TPassivePlayer.OpenPassiveConnection}
 6:     WITH fADSP^ DO BEGIN
 7:         ioCompletion    := NIL;
 8:         filterAddress   := AddrBlock(0);
 9:         ocMode          := ocPassive;
10:         ocInterval      := 0;
11:         ocMaximum       := 0;
12:         csCode          := dspOpen;
13:     END;
14:     stat := PBControl(ParmBlkPtr(fADSP),kASYNC);
15:     fPlayerState := kConnecting;
16:     DoOperation;
17: END; {TPassivePlayer.OpenPassiveConnection}
```

Line 15 sets the player state to connecting and line 20 calls **DoOperation** to start the polling. This uses the **TAsyncOp** technique for handling asynchronous operations during idle times.

TActivePlayer.DoAfterConnection

The **DoAfterConnection** method is called after the **dspOpen** completes. This allows the active player to set up the game as it should be at the beginning of play. Listing 12-33 displays the **DoAfterConnection** method.

Listing 12-33 The DoAfterConnection method (TActivePlayer.DoAfterConnection)

```
{$S ARes}
PROCEDURE TActivePlayer.DoAfterConnection; OVERRIDE;
BEGIN {TActivePlayer.DoAfterConnection}
    fPlayerState := kMyTurn;
    MyTurn(kPlaying);;
END; {TActivePlayer.DoAfterConnection}
```

Line 4 sets the player state to be this player's turn. The active player always goes first.

Line 5 calls the **MyTurn** procedure that tells the checkers application that it should start this player's turn.

TPassivePlayer.DoAfterConnection

The **DoAfterConnection** method is called after the **dspOpen** completes. This allows the passive player to set up the game as it should be at the beginning of play. Listing 12-34 shows the **DoAfterConnection** method.

Listing 12-34. The DoAfterConnection method (TPassivePlayer.DoAfterConnection)

```
1: {$S ARes}
2: PROCEDURE TPassivePlayer.DoAfterConnection; OVERRIDE;
3: BEGIN {TPassivePlayer.DoAfterConnection}
4:     fPlayerState := kYourTurn;
5:     ReadAMessage;
6: END; {TPassivePlayer.DoAfterConnection}
```

Line 4 sets the player state to be the other player's turn. The passive player always goes second.

Line 5 calls the **ReadAMessage** method to get ready for an incoming message from the active player.

▶ Summary

This chapter described a series of Pascal object classes that implement the communications needed to build a two-player checkers game. The code illustrated asynchronous operations using polling during idle times. It also showed how to set up an ADSP connection and then send messages back and forth over that connection.

Index

Macintosh Programming/Networking

Macintosh Inside Out

>$24.95 USA
>$31.95 CANADA

Programming with AppleTalk®

MICHAEL PEIRCE

AppleTalk® is the comprehensive network system built into every Macintosh®. All Macintosh programmers, beginners or advanced, need to learn the ins and outs of working with the AppleTalk system.

Programming with AppleTalk is the first hands-on guide to understanding and working with AppleTalk. This book describes the important features and functions of the system in detail, showing you how to create applications and system extensions that run with AppleTalk. Topics covered include AppleTalk protocols and the protocol stack, transport media, the Preferred AppleTalk Interface, and storage management. Numerous working c examples walk you through using RDEV, INIT, NBP, ATP, and ADSP and provide special tips and techniques for writing efficient and effective programs.

You will also learn how to:

- Use synchronous and asynchronous calls
- Avoid heap fragmentation
- Configure a Chooser Interface
- Work with AppleTalk drivers

and much more.

This thorough coverage of vital AppleTalk concepts and features makes **Programming with AppleTalk** an essential reference for all Macintosh programmers.

Michael Peirce is an independent computer consultant specializing in Macintosh networking. He formerly worked as a [illegible] software engineer with Claris Corporation, where he created a popular network file sharing utility, Public Folder.

X000PS4BSF
Programming with AppleTal
Used Good
UsedGood L1A14
2015-01-13
E1776C3
CAE1

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.

ISBN 0-201-57780-1
57780