This chapter describes how you can use the Datagram Delivery Protocol (DDP) to send data to and receive it from another socket across an AppleTalk internet. To use DDP, you send and receive data as discrete packets, each packet carrying its own addressing information. DDP does not allow you to set up a connection between two sockets, nor does DDP ensure that data is delivered error free as do some of the AppleTalk protocols that are built on top of it.

You should use DDP if your application does not require reliable delivery of data and you do not want to incur the additional processing associated with the use of a protocol that entails setting up and breaking down a connection. Because it is connectionless and does not include reliability services, DDP offers faster performance than do the higher-level protocols that add these services. Applications such as diagnostic tools that retransmit packets at regular intervals to estimate averages or games that can tolerate packet loss are good candidates for the use of DDP.

A series of DDP packets transmitted over an AppleTalk internet from one node to another may traverse a single high-speed EtherTalk network or they may wind across multiple intermediate data links such as LocalTalk or TokenTalk, which are connected by routers. During the course of this process, some packet loss can occur, for example, as a result of collisions. If you do not plan on implementing recovery from packet loss in your application, but your application requires it, you should consider using an AppleTalk transport protocol, such as the AppleTalk Data Stream Protocol (ADSP) or the AppleTalk Transaction Protocol (ATP); these protocols protect against packet loss and ensure reliability by using positive acknowledgment with packet retransmission mechanisms.

This chapter describes how to

- open and close sockets for sending and receiving DDP packets
- prepare the data and addressing information for each packet that you want to send
- write a socket listener that receives packets addressed to the DDP socket associated with your application
- measure packet-delivery performance

This chapter includes a sample socket listener that you can use as a model for your own socket listener or modify to fit your application's requirements.

For an overview of DDP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter.

For an explanation of the DDP specification, see *Inside AppleTalk*, second edition.

About DDP

The protocol implementations at the physical and data-link layers of the AppleTalk protocol stack provide node-to-node delivery of data on the internet. DDP is a client of the link-access protocol—whether LLAP, ELAP, TLAP, or FDDILAP—and it uses the node-to-node delivery services provided by the data link to send and receive data. DDP is responsible for delivering data from socket to socket over an AppleTalk internet.

DDP is central to the process of sending and receiving data across an AppleTalk internet. Regardless of which data link is being used and which (if any) higher-level protocols are processing data, all AppleTalk data is carried in the form of DDP packets known as ***datagrams.*** (This chapter uses the terms *datagram* and *DDP packet* interchangeably.) A datagram consists of a header followed by data.

DDP lets you send and receive data a packet at a time. If you use DDP, you must address each data packet to the ***socket*** for which it is intended. A socket is a piece of software that serves as an addressable entity in a networked node. Sockets are numbered, and each application that uses DDP to transfer data is associated with a unique socket. You cannot open and maintain a session between two sockets using DDP, and for this reason, DDP is called a *connectionless protocol.*

To use DDP, you must provide a ***socket listener*** and a routine that reads packets from the socket listener code after it receives them. A socket listener is a process that receives packets addressed to the DDP socket associated with your application. Because the driver that implements DDP, the .MPP driver, uses registers not accessible from higher-level languages such as Pascal to pass information to your socket listener, you must write the socket listener code in assembly language.

DDP is said to provide a best-effort socket-to-socket delivery of datagrams over the internet.

- *Socket-to-socket delivery* means that when the data link delivers a packet to a node, the DDP implementation in that node determines the socket for which the packet is intended and calls the socket listener for that socket.
- *Best-effort delivery* means that DDP attempts to deliver any datagram that has a valid address to an open socket, as long as the length of the datagram received is the same as the length indicated by the header, the data is not longer than 586 bytes, and the datagram does not include an invalid checksum. DDP has no provision for requesting the sender to retransmit a lost or damaged datagram.

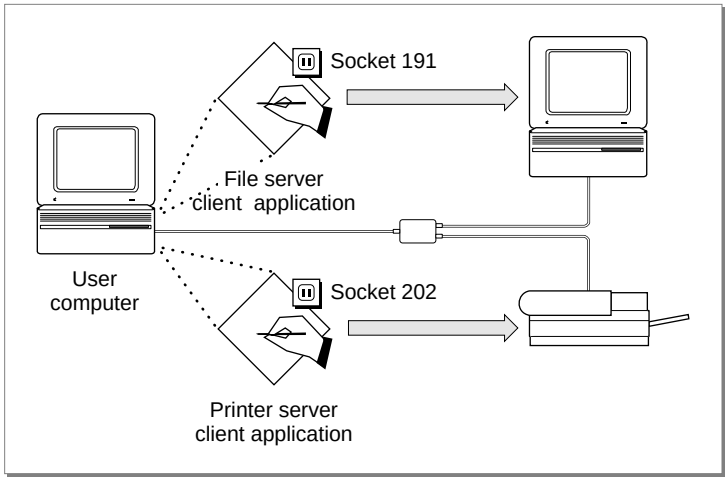
Note
You can send DDP packets to another socket in your own node if you have enabled the intranode delivery feature of AppleTalk. By default, intranode delivery is disabled; to turn it on, you use the `PSetSelfSend` function, which is described in the chapter "AppleTalk Utilities" in this book. ◆

About Sockets and Socket Listeners

Every application that uses DDP to transfer data must send or receive that data through a socket. The use of sockets allows DDP to determine for which application a packet is intended. Each node supports up to 254 sockets, and each socket is identified by an 8-bit number that combines with the network number and the node ID to form the internet socket address of the application. When an application or process calls DDP to open a socket, DDP associates the number of that socket with the application, making the application distinct from other applications on the same node. An application that is associated with a specific socket through DDP is the client of that socket, or a ***socket client.***

The use of sockets allows multiple processes or applications that run on a single node connected to AppleTalk to be open at the same time. In Figure 7-1, a printer server client application and a file server client application are open on the same node at the same time. Each application is associated with a unique socket, and packets for that application are addressed to that socket number.

Figure 7-1 Two applications running on the same node, each with its own socket



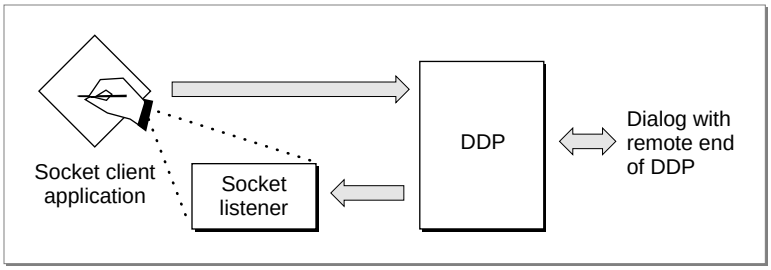
Applications exchange data with each other through their sockets. A socket client can send and receive datagrams only through its associated socket. Moreover, every socket-client application that uses DDP directly to transfer data must have associated with it a socket listener that receives datagrams addressed to the socket on behalf of that socket's client application.

A socket listener is a process that you provide as part of your client application. You must write your socket listener in assembly language and adhere to specific requirements in regard to the use of registers and the routines that you call to receive packets. Beyond meeting these AppleTalk requirements, your socket listener can perform any other functions that your socket-client application requires. See "A Sample Socket Listener" beginning on page 7-20 for more details.

When you call DDP to open a socket, you provide a pointer to your socket listener for that socket. DDP maintains a ***socket table*** that includes an entry for every open socket and its socket listener. When the .MPP driver receives a packet, it does not read and process the packet. Instead, it reads the socket number portion of the internet socket address and then checks the socket table to determine if that socket is open. If so, the .MPP driver calls the socket listener associated with the socket to handle reception of the packet for the client application. The use of socket listeners helps to maximize throughput between DDP and the link-access protocol layer by eliminating unnecessary buffer copying.

Figure 7-2 shows a socket-client application that calls DDP to send data to another socket. The socket-client application includes code that comprises its socket listener. When DDP receives a packet addressed to this socket, it checks the socket table for the entry that contains the socket number and the address of the socket listener belonging to the application that owns the socket; then DDP calls the socket listener to receive the packet for the application.

Figure 7-2 Sending and receiving data using DDP



Assigning Socket Numbers

DDP maintains two classes of sockets: sockets that are assigned statically and sockets that are assigned dynamically. There are some restrictions on which socket numbers they use:

- Statically assigned sockets have numbers in the range of 1–127.
  - Socket numbers 1–63 are reserved for use by Apple Computer, Inc.
  - Socket numbers 64–127 are available for program development.
- Dynamically assigned sockets have numbers within the range of 128–254.

To use a statically assigned socket, an application must request a specific socket number. In most cases, you should not use statically assigned sockets.

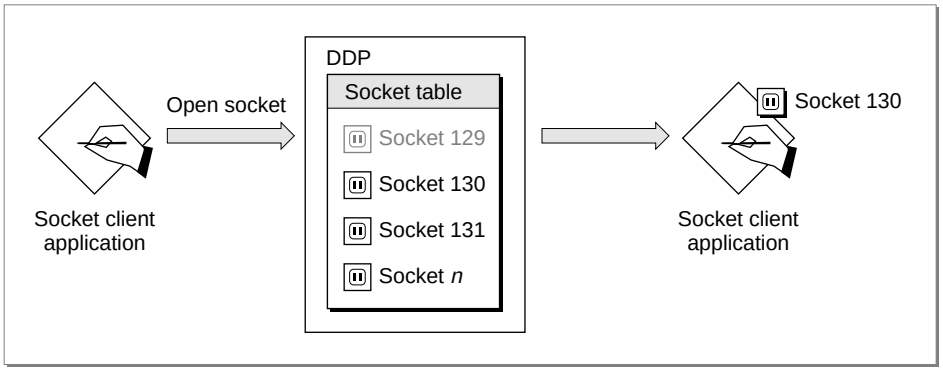
IMPORTANT

Although you can use statically assigned sockets whose numbers fall within the range of 64–127 for program development, you must *not* use a statically assigned socket number for a released product. To do so creates the possibility of conflicts arising, for example, when two applications that both use the same statically assigned socket are open on the same node at the same time. Data intended for one application could be delivered to the other application, and vice versa. ▲

DDP maintains a pool of available sockets from which it selects a socket number to assign dynamically for your use when you call DDP to open a socket and you do not specify a number within the range of statically assigned sockets.

Figure 7-3 illustrates conceptually what happens when an application calls DDP to open and assign a socket dynamically. In this example, DDP assigns socket number 130 to the application that requests a socket. (Socket number 129 is already assigned to an application.)

Figure 7-3 Assigning sockets



To let DDP choose a socket number from the pool of available sockets within the range of dynamically assigned sockets, you specify 0 for the socket number. However, you can choose a specific socket within that range and pass the number of that socket to DDP to open. If that socket is available, DDP opens it, assigns it to your application, and associates your socket listener with it. If the socket number you specify is not available, DDP returns an error result.

DDP Client Protocol Types

AppleTalk allows for the implementation of up to 254 parallel protocols that are clients of DDP. The DDP protocol type field, which is the last field of the DDP packet header, specifies the type of protocol that the packet is intended for. Figure 7-6 on page 7-15 shows the 1-byte DDP protocol type field of the DDP packet header.

The socket listener for a single socket can receive packets whose protocol type fields contain different values. It is the responsibility of your socket-client application to define its own protocol types. Your socket-client application can define more than one DDP protocol type and receive packets for any of the protocol types it handles, sorting them by reading the value of the DDP protocol type field.

For example, if you are implementing a server, you might define one protocol type for data and another for attention messages, and have separate routines to handle the different packet types. You fill in the DDP protocol type field when you build the contents of a DDP packet to be sent to another socket.

For more information on how to specify a protocol type for a DDP client application and the range of valid values for the DDP protocol type field, see Appendix C in *Inside AppleTalk*, second edition.

Obtaining Data From the Network

DDP supports a number of client protocols that are built on top of it, and DDP itself is a client protocol of the underlying data-link protocol. DDP has its own protocol handler that the link-access protocol calls when it receives a DDP packet. A ***protocol handler*** is a process that receives packets for a specific protocol type much like a socket listener receives packets for a specific socket. A DDP packet or datagram is sent from its source socket through one or more AppleTalk networks to its destination network.

A datagram is sent across the network enclosed in a ***frame.*** The frame contains additional information that the link-access protocol requires, such as addressing information that identifies the node and the socket number for which the frame is meant. The frame addressing information is contained in the frame's header, which is followed by the datagram. The frame header also identifies the protocol type of the enclosed packet. In addition to a header, a frame also contains a trailer that follows the datagram. The frame trailer contains a frame check sequence number that the AppleTalk hardware generates and uses to detect transmission errors.

The link-access protocol in the destination network delivers the frame to the node containing the destination socket. When a frame addressed to a particular node arrives at that node, the node's CPU is interrupted and the .MPP driver's interrupt handler gets control to service the interrupt. As the frame's first 3 bytes are read into the first-in first-out (FIFO) buffer, the .MPP driver's interrupt handler moves these bytes into its own internal buffer.

If the frame is a data frame containing a packet intended for a higher-level protocol, the .MPP driver's interrupt handler passes control to the protocol handler for the protocol type specified in the frame's header. For example, when a frame whose header specifies the DDP protocol type is delivered to a node, the link-access protocol passes control to the .MPP driver. The .MPP driver then calls the DDP protocol handler. DDP, which is implemented by the .MPP driver, determines for which socket the packet is meant and calls the socket listener that is associated with the socket. The socket listener, in turn, actually reads in the packet.

Using DDP

This section describes how to send data packets to a socket and how to receive them from another socket over an AppleTalk network or internet using DDP. It also describes how to use the AEP Echoer to measure packet-delivery performance and to determine if a node is on the network.

Note
You do not need to use the AEP Echoer to send and receive data using DDP. This chapter describes the AEP Echoer because you must use the programming interface to DDP in order to use the AEP Echoer. Applications that use higher-level AppleTalk protocols, such as ATP or ADSP, can also use the AEP Echoer to measure packet-delivery performance. ◆

Sending and Receiving Data: An Overview

To send data, you must address each packet to the socket for which it is intended because you cannot open and maintain a connection between two sockets using DDP. To receive a data packet using DDP, you must provide a socket listener process that DDP associates with the socket that your application uses. When you open the socket for your application to use, you must provide a pointer to the socket listener. DDP associates the address of the socket listener with your application's socket so that the .MPP driver can call your socket listener when it receives a packet that is addressed to your socket-client application. DDP maintains a separate entry in its socket table for each socket and socket listener pair.

Applications developers commonly write a single socket-client application that both sends and receives data and that includes a socket listener process to receive data. To clarify the steps involved in sending and receiving data, this section gives you an overview of these tasks as separate sequences after it explains how to open a socket. The steps for sending and receiving data refer to sections that are provided later in this chapter that describe how to

- create a write-data structure, which you need to send data
- use the registers that the .MPP driver uses to pass parameters to your socket listener
- write a socket listener, with sample code illustrating this

If you want to provide features in addition to the DDP checksum feature to check data and correct errors, you can include them in your application, you can define your own AppleTalk protocol, or you can use a higher-level AppleTalk protocol such as ATP or ADSP instead of calling DDP directly. (For information about DDP checksums, see "Using Checksums" beginning on page 7-19.)

To make your application available to other users of AppleTalk, you must use the NBP `PRegisterName` function to register the name that represents your socket-client application. When you are finished using the socket, you must use the NBP `PRemoveName` function to remove this name from the NBP names table. See the chapter "Name-Binding Protocol (NBP)" in this book for more information about these functions.

Opening a Socket

To send and receive data using DDP, your application must first open a socket. Opening a socket makes your application a client of that socket. You open a socket with the `POpenSkt` function. When you open a socket, you must provide a pointer to your socket listener and you must specify 0 for the socket number if you want DDP to dynamically assign a socket.

The `POpenSkt` function assigns a socket number to your application and enters the number in the socket table along with the pointer to the socket listener that you provide. The `POpenSkt` function returns the socket number to you in the `socket` field of the parameter block.

Associating a single socket listener with more than one socket

If your application includes processes that each have their own sockets, you can assign a single socket listener to more than one socket, but each socket should have its own buffer or set of buffers for receiving data. ◆

If you do not want DDP to randomly assign a socket number to your application, you can specify the number of a particular socket for DDP to open. For information on the range of socket numbers from which you can select, see "Assigning Socket Numbers" on page 7-6.

IMPORTANT

You cannot specify a `NIL` pointer to the socket listener. If you do, the system on which your application is running will crash. ▲

When your application is finished using a socket, you must use the `PCloseSkt` function to close the socket.

Sending Data

To send data, you must create a write-data structure that contains the data in a specific format and then call a DDP function to send the data. After you have opened a socket using the `POpenSkt` function, here are the steps that you follow to send data using DDP:

1. Create a write-data structure.
2. Use the DDP function `PWriteDDP` to send the data.

See "Creating a DDP Write-Data Structure" beginning on page 7-12 for information about how to create a write-data structure using the DDP procedure `BuildDDPwds` or your own code.

Packets with long headers can include a checksum that can be used to verify the integrity of the packet data. For information on how to direct DDP to calculate a checksum for data that you want to send, see "Using Checksums" beginning on page 7-19. For details of the contents of a long header, see "The DDP Packet and Frame Headers" beginning on page 7-14.

Receiving Data

To receive data using DDP, you must provide a socket listener that is part of your socket-client application. The socket listener code must

- be written in assembly language because it must read from and write to the CPU's registers
- include buffers to hold the data that it reads
- use the register values that the .MPP driver passes to your socket listener
- determine the type of packet, if you have defined more than one protocol type that your application handles
- if the packet includes a long header, calculate the checksum value, if one is used

There are many ways to design and write a socket-client application and socket listener. This chapter offers one possibility. For details of this sample socket listener and for its code, see "A Sample Socket Listener" beginning on page 7-20.

Note
Your socket-client application should test to find out when the socket listener finishes processing a packet so that the socket-client application can begin its own packet reading and processing. ◆

To receive data, your application must have already opened a socket using the `POpenSkt` function and have passed the `POpenSkt` function a pointer to your socket listener.

Here are the tasks involved in receiving data using DDP:

1. The .MPP driver calls your socket listener when it receives a packet addressed to your socket-client application. The .MPP driver passes values to you in the CPU's registers. You need to know how the .MPP driver uses these registers and how you can use them. For information about these registers, see "How the .MPP Driver Calls Your Socket Listener" beginning on page 7-13. One of the values that the .MPP driver passes to you is a pointer to the buffer that holds the DDP packet header. You need to know how the DDP packet header and the frame header are structured. For information about these headers, see "The DDP Packet and Frame Headers" beginning on page 7-14.
2. To hold the data that it reads, your socket listener must allocate memory for buffers. In addition to allocating data buffers, either your socket-client application or the socket listener (if you write the socket listener code to carry out this function) must perform some initialization tasks. For information about these tasks and how the sample socket listener handles them, see "Socket Listener Queues and Buffers" beginning on page 7-20, "Setting Up the Socket Listener" beginning on page 7-22, and "Initializing the Socket Listener" beginning on page 7-24.
3. When the .MPP driver calls your socket listener, the socket listener must read the incoming packet into one or more data buffers. To do this, the socket listener uses two processes, `ReadPacket` and `ReadRest`, which are implemented as a single routine in the hardware driver. The .MPP driver passes you the address of this routine in one of the CPU's registers. For more information, see "Reading an Incoming Packet" beginning on page 7-17.
4. If you have defined more than one DDP protocol type that your application handles, check the DDP protocol type field of the datagram header (see Figure 7-6 on page 7-15) to determine the protocol type of the packet you have just received.

   The AppleTalk internet address (network number, node ID, and socket number) is insufficient to distinguish between packets intended for different processes that are using the same socket. Your socket listener must use some other information (such as the DDP protocol type or a higher-level protocol header imbedded in the DDP packet data) to make this distinction.
5. If the packet contains a long header, the socket listener needs to find out if the header contains a checksum. If it does, the socket listener needs to calculate the checksum to determine if the packet's data has been corrupted. For more information, see "Using Checksums" beginning on page 7-19.

6. The socket listener can now process the packet or pass it to the client application for processing. The sample socket listener provided here writes the packet buffer to a queue that it uses for successfully processed packets and removes the packet from the queue for incoming packets. For a description of how the sample socket listener does this, see "Processing a Packet" beginning on page 7-25.
7. The client application can now read in the packet for its own purposes. The client application should include code that periodically checks to determine whether the socket listener has finished processing an incoming packet. For a description of how the sample socket listener's client application performs this task and some sample code, see "Testing for Available Packets" beginning on page 7-31.

Creating a DDP Write-Data Structure

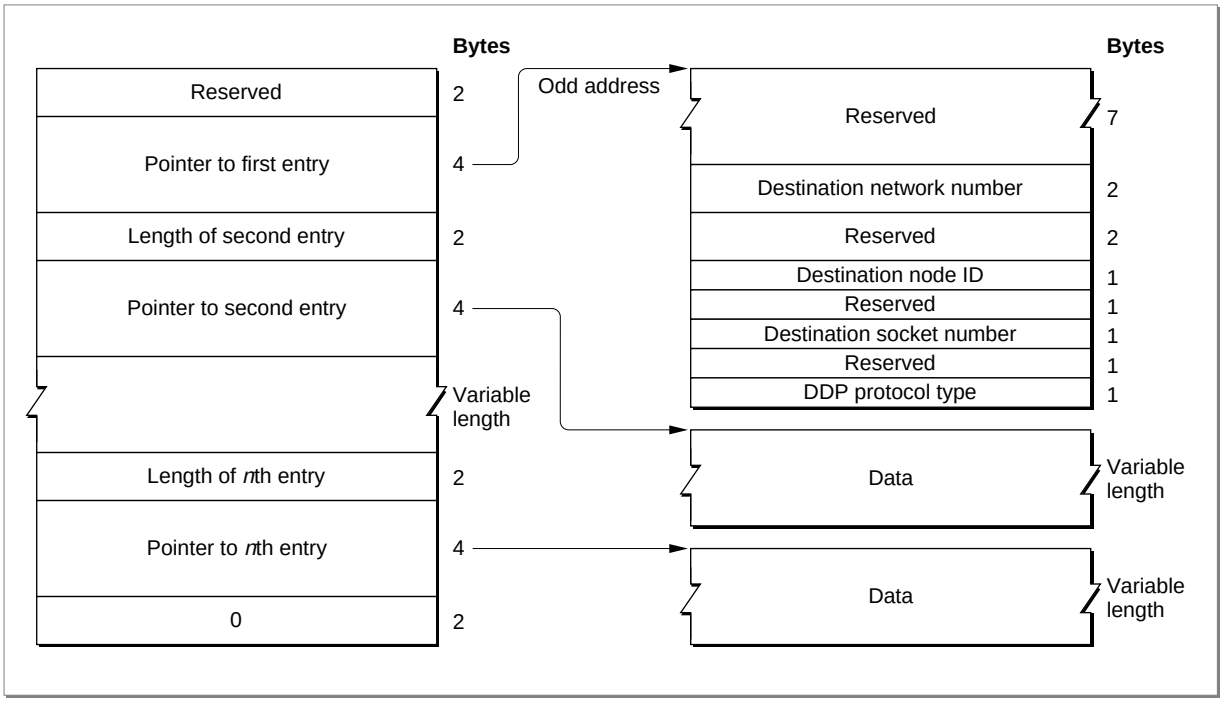
When you use the `PWriteDDP` function to send a DDP packet to another socket, you provide a pointer to a write-data structure that you have already created. A write-data structure contains a series of pairs of length words and pointers and ends with a 0 word. Each pair indicates the length and location of a portion of the data that constitutes the packet to be sent over the network. The first entry in the write-data structure consists of only a pointer. It does not include a length word, because the length is always the same.

The first pointer indicates a 16-byte header block, which must start at an odd address. You fill in the destination network number, destination node ID, destination socket number, and DDP protocol type, and the .MPP driver fills in the other fields of the packet header. DDP protocol types 1 through 15 are reserved for use by Apple. A DDP packet may have a maximum of 586 bytes of data. Figure 7-4 shows the write-data structure and the header block.

Because the first pointer in the write-data structure must point to an odd address, it is difficult to use Pascal to create a write-data structure. If you are programming in Pascal, you can use the `BuildDDPwds` procedure to create a write-data structure. You must provide a 17-byte buffer for the header block, a 14-byte buffer to hold the write-data structure, and a pointer to the data you want to send. The header block is only 16 bytes, but because it begins on an odd address, the first byte is not used. The write-data structure created by the `BuildDDPwds` procedure is 14 bytes long, consisting of only a pointer to the header, a length-pointer pair for the data block, and the terminating 0 word. Although a write-data structure allows you to divide the data into as many blocks as you wish, the `BuildDDPwds` procedure assumes that the data is in a single block.

In most cases, if you are using DDP directly to send data across a network, a single block of data should be adequate. However, if you are implementing a protocol on top of DDP and you want to send blocks of data that are stored separately as parts of the same datagram, you will have to build your own write-data structure that includes multiple pairs of pointers and lengths. For a description of the write-data structure that you need to build in this case, see "The Write-Data Structure" on page 7-35. Notice that the pointer to the first entry indicates an odd address and that there is no length word for the first entry.

Figure 7-4 DDP write-data structure



Using Registers and Packet Headers

To receive data at the DDP level, you need to include as part of your socket-client application a socket listener that reads packets addressed to your application and passes them to the application for further processing. DDP maintains a table with an entry for each socket and socket listener pair.

When the .MPP driver receives a packet addressed to your socket-client application, it calls your socket listener, using the CPU's registers to pass pointers to the internal buffer where it has stored the packet's headers and to some data values that your socket listener uses during its processing.

The CPU's registers that the .MPP driver uses to pass parameters to your socket listener are not directly accessible from Pascal. Because a DDP socket listener must read from and write to the CPU's registers, you must write a socket listener in assembly language; you cannot use Pascal. However, you can write the remainder of the client application that includes the socket listener in a high-level language such as Pascal. The client application sample code that this chapter shows is written in the Pascal language.

How the .MPP Driver Calls Your Socket Listener

When a frame addressed to a particular node arrives at that node and the frame contains a DDP packet, the node's CPU is interrupted and the link-access protocol calls the .MPP driver to receive the packet. When the .MPP driver receives a DDP packet, it reads the

first 3 bytes of the frame header into an internal buffer called the ***read-header area (RHA).*** After the frame header is read into the RHA, 8 bytes of the RHA are still available for your use.

Next, the .MPP driver reads the socket address and calls the socket listener for that socket. The .MPP driver uses the CPU's registers to pass parameters to your socket listener as follows:

Registers on call to DDP socket listener

A0 Reserved for internal use by the .MPP driver. You must preserve this register until after the `ReadRest` routine has completed execution.

A1 Reserved for internal use by the .MPP driver. You must preserve this register until after the `ReadRest` routine has completed execution.

A2 Pointer to the .MPP driver's local variables. The value at the offset `toRHA` from the value in the A2 register points to the start of the RHA.

A3 Pointer to the first byte in the RHA past the DDP header bytes (the first byte after the DDP protocol type field).

A4 Pointer to the `ReadPacket` routine. The `ReadRest` routine starts 2 bytes after the start of the `ReadPacket` routine.

A5 Free for your use before and until your socket listener calls the `ReadRest` routine.

D0 Lower byte is the destination socket number of the packet.

D1 Word indicating the number of bytes in the DDP packet left to be read (that is, the number of bytes following the DDP header).

D2 Free for your use.

D3 Free for your use.

When the .MPP driver calls your socket listener, you can read the destination socket number that is in the D0 register and the frame header that is in the RHA. You should assume that only 8 bytes are still available in the RHA for your use. Figure 7-5 shows the beginning of the RHA where the frame header begins; the frame header is followed by either a short or a long DDP header.

The DDP Packet and Frame Headers

A DDP packet includes a packet header followed by data. The DDP packet header is preceded by the frame header. Figure 7-6 shows both headers; they do not include the data portion. The DDP packet header can be long or short; if the destination and source network numbers are different, DDP uses a long header, which includes some additional fields.

The frame header includes

- the source and destination node IDs
- the DDP header type (1 = short, 2 = long)

Figure 7-5 The RHA for both long and short DDP headers

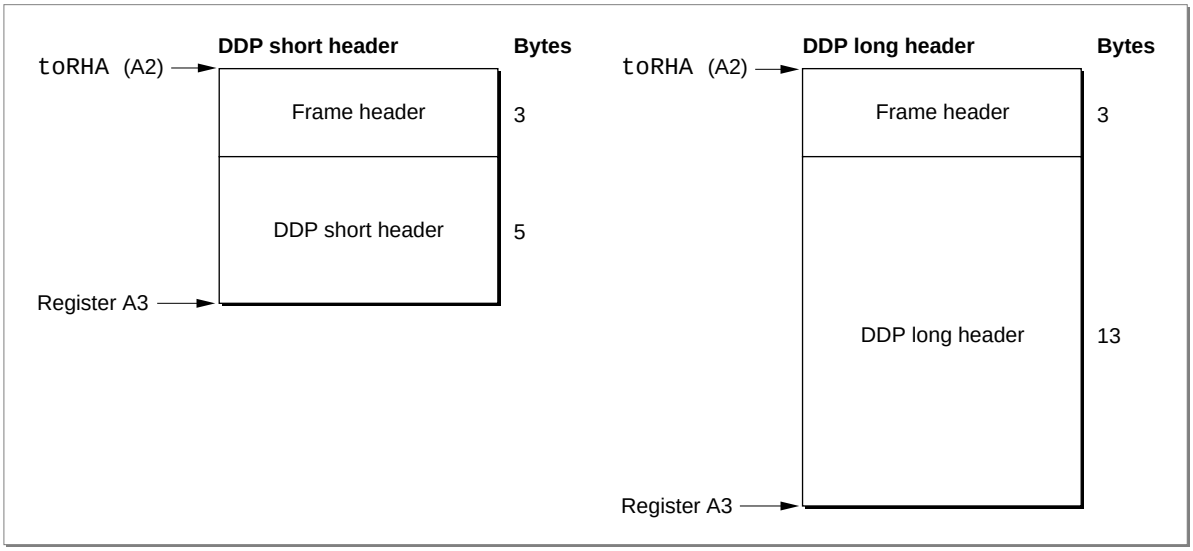
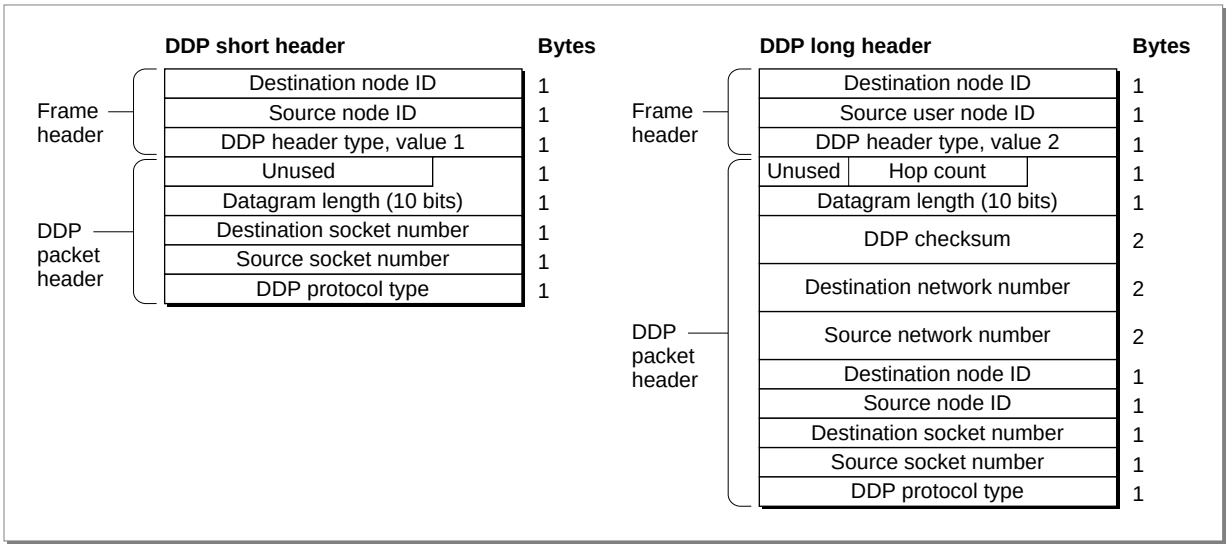


Figure 7-6 Data-link frame header and DDP packet header



The DDP long and short packet headers have these fields in common:

- the datagram length (10 bits)
- the destination socket number
- the source socket number
- the DDP protocol type

A long DDP packet header also includes

- a hop count
- a checksum value, if one was calculated
- the source network number and node ID
- the destination network number and node ID

The MPW Equates

You can use the following equates from the MPW interface files in writing your socket listener process and the client application that includes it:

```
;frame header
;
lapDstAdr   EQU   0     ;destination node address [byte]
lapSrcAdr   EQU   1     ;source node address [byte]
lapType     EQU   2     ;LAP type field [byte]
lapHdSz     EQU   3     ;size of LAP header

;DDP packet header
;
ddpHopCnt   EQU   0     ;hop count (only used in long
                        ; header) [byte]
ddpLength   EQU   0     ;packet length (from this word
                        ; onward) [word]
ddpChecksum EQU   2     ;checksum [word]
ddpDstNet   EQU   4     ;destination network no. [word]
ddpSrcNet   EQU   6     ;network of origin [word]
ddpDstNode  EQU   8     ;destination node address [byte]
ddpSrcNode  EQU   9     ;node of origin [byte]
ddpDstSkt   EQU   10    ;destination socket number [byte]
ddpSrcSkt   EQU   11    ;source socket number [byte]
ddpType     EQU   12    ;DDP protocol type field [byte]
sddpDstSkt  EQU   2     ;destination socket number (short
                        ; header) [byte]
sddpSrcSkt  EQU   3     ;source socket number (short
                        ; header) [byte]
sddpType    EQU   4     ;DDP protocol type field (short header)
                        ; [byte]
;
ddphSzLong  EQU   13    ;size of extended DDP header
ddphSzShort EQU   5     ;size of short DDP header
;
shortDDP    EQU   $01   ;LAP type code for DDP (short header)
longDDP     EQU   $02   ;LAP type code for DDP (long header)
```

Reading an Incoming Packet

Your socket listener calls the `ReadPacket` and `ReadRest` processes to read the incoming data packet. You can call `ReadPacket` as many times as you like to read the data piece by piece into one or more data buffers, but you must always use `ReadRest` to read the final piece of the data packet. Alternatively, you can read all of the data using only `ReadRest`. The `ReadRest` routine restores the machine state (the stack pointers, status register, and so forth) and checks for error conditions.

Note
You can ignore any remaining data instead of reading it by setting the D3 register to 0 and calling `ReadRest`. ◆

Before you call the `ReadPacket` routine, you must allocate memory for a data buffer and place a pointer to the buffer in the A3 register. You must also place the number of bytes you want to read in the D3 register. You must not request more bytes than remain in the data packet.

The buffer that you allocate must be large enough to hold all of the data and—if your socket listener places the packet header in the buffer—the header as well. The maximum amount of data in a DDP data packet is 586 bytes. A long DDP packet header is 13 bytes long; a short header is 5 bytes. The frame header is 3 bytes. Therefore, the maximum amount of data from the packet that the socket listener can return is 602 bytes. You can use the buffer as a data structure to hold other information as well, such as the number of bytes of data actually read by the socket listener, a flag that indicates when the data has been returned, and result codes.

After you have called the `ReadRest` routine, you can use registers A0 through A3 and D0 through D3 for your own use, but you must preserve all other registers. You cannot depend on having access to your application's global variables.

To call the `ReadPacket` routine, execute a JSR instruction to the address in the A4 register. The `ReadPacket` routine uses the registers as follows:

Registers on entry to the `ReadPacket` routine

A3 Pointer to a buffer to hold the data you want to read

D3 Number of bytes to read; must be nonzero

Registers on exit from the `ReadPacket` routine

A0 Unchanged

A1 Unchanged

A2 Unchanged

A3 Address of the first byte after the last byte read into buffer

A4 Unchanged

D0 Changed

D1 Number of bytes left to be read

D2 Unchanged

D3 Equals 0 if requested number of bytes were read, nonzero if error

After every time you call `ReadPacket` or `ReadRest`, you must check the zero (`z`) flag in the status register for errors because the `ReadPacket` routine indicates an error by clearing it to 0. If the `ReadPacket` routine returns an error, you must terminate execution of your socket listener with an RTS instruction without calling `ReadPacket` again or calling `ReadRest` at all.

Call the `ReadRest` routine to read the last portion of the data packet, or call it after you have read all the data with `ReadPacket` routines and before you do any other processing or terminate execution. After you call the `ReadRest` routine, you must terminate execution of your socket listener with an RTS instruction whether or not the `ReadRest` routine returns an error.

When you call the `ReadRest` routine, you must provide in the A3 register a pointer to a data buffer and must indicate in the D3 register the size of the data buffer. If you have already read all of the data with calls to the `ReadPacket` routine, specify a buffer of size 0.

▲ **WARNING**
If you do not call the `ReadRest` routine after the last time you call the `ReadPacket` routine successfully, the system will crash. You do not need to call the `ReadPacket` routine; you can call only the `ReadRest` routine to read in the entire packet. However, you must call the `ReadRest` routine. ▲

To call the `ReadRest` routine, execute a JSR instruction to an address 2 bytes past the address in the A4 register. The `ReadRest` routine uses the registers as follows:

Registers on entry to the `ReadRest` routine

A3 Pointer to a buffer to hold the data you want to read

D3 Size of the buffer (word length); may be 0

Registers on exit from the `ReadRest` routine

A0 Unchanged

A1 Unchanged

A2 Unchanged

A3 Pointer to first byte after the last byte read into buffer

D0 Changed

D1 Changed

D2 Unchanged

D3 Equals 0 if requested number of bytes exactly equaled the size of the buffer; less than 0 if more data was left than would fit in buffer (extra data equals –D3 bytes); greater than 0 if less data was left than the size of the buffer (extra buffer space equals D3 bytes)

Calling `ReadPacket` and `ReadRest` when LocalTalk is the data link

If LocalTalk is the data link that is being used, your socket listener has less than 95 microseconds (best case) to read more data with a `ReadPacket` or `ReadRest` call. If you need more time, you can read another 3 bytes into the RHA, which will allow you an additional 95 microseconds. ◆

In implementing your socket listener, you can use the registers as follows:

- You can use registers D0, D2, and D3 freely throughout the socket listener code.
- You must preserve the contents of registers A6 and D4 to D7.
- From entry to your socket listener until you call `ReadRest`
  - you can use A5 register
  - you must preserve registers A0 to A2, A4, and D1
- From `ReadRest` until your application exits from the socket listener
  - you must preserve register A5
  - you can use registers A0 to A3 and D0 to D3

Using Checksums

For packets that include a long header, DDP includes a checksum feature that you can use to verify that the packet data has not been corrupted by memory or data bus errors within routers on the internet.

When you use the `PWriteDDP` function to send a DDP packet across an AppleTalk internet, you can set a flag (`checksumFlag`) to direct DDP to calculate a checksum for the packet.

If the checksum flag is set and the socket to which you are sending the packet (the destination socket) has a network number that is different from that of the socket from which you are sending the packet (the source socket), then the `PWriteDDP` function calculates a checksum for the datagram and includes it in the datagram packet header. In this case, DDP uses a long header for the packet; Figure 7-6 on page 7-15 shows both the long and short DDP headers.

When your socket listener receives a packet that has a long header, the socket listener must determine whether DDP calculated a checksum for the packet, and if so, use the checksum to verify that the data was delivered intact. You can use the equates from the MPW interface files in calculating checksums: see "The MPW Equates" on page 7-16.

To determine this, your socket listener code should take the following steps:

1. Check the DDP header type field. This is set to 2 for a packet with a long header and 1 for a packet with a short header.
2. Check the checksum field (`checksumFlag`). This is set to a nonzero value if the sender specified that DDP should calculate a checksum for the packet; a short header does not include a checksum field.

3. Calculate the checksum using the following algorithm to calculate the checksum, starting with the byte immediately following the checksum field in the header and ending with the last data byte:

   checksum := checksum + next byte; {unsigned addition}
   Rotate the most significant bit to the least significant bit
   Repeat

4. Compare the calculated checksum against the value set in the checksum field of the DDP packet header.

   You can use the equates from the MPW interface files in calculating checksums: see "The MPW Equates" on page 7-16.

A Sample Socket Listener

There are many ways to implement a socket listener that follow the requirements described previously for using and preserving registers and reading packets. This section uses a sample socket listener that shows one way to implement the process within a DDP socket-client application that reads in the packet contents. The sample code also shows those segments of the sample client application that set up the socket listener and check to determine when a packet that the socket listener has read is available for processing by the client application.

Some of the tasks that your socket listener can do that this sample socket listener does not illustrate are how to

- route packets to different sockets based on the socket number in register D0 when more than one socket uses your socket listener
- check the DDP protocol type field and ignore any packets that do not match the desired packet types that your socket listener is set up to receive
- check the source node ID and ignore any packets that don't come from a desired node
- implement a completion routine to be executed after a packet is processed

The sample socket listener does, however, show you how to

- buffer multiple packets
- retrieve the frame and DDP packet header information that DDP has already read into the RHA
- calculate and compare the packet checksum when a packet uses a long DDP header that includes the checksum value

Socket Listener Queues and Buffers

The sample socket listener uses two standard operating-system queues to manage the contents of the packets that it receives and makes available to the socket-client application. It calls these linked lists a *free queue* and a *used queue*. The use of two queues allows the socket listener to receive and process packets while the client application is reading the data from those packets that the socket listener has already processed.

The free queue is used to manage available buffers that consist of data structures declared as `PacketBuffer` records. The sample socket listener uses the buffers in the free queue one at a time to hold the contents of an incoming packet as it processes the packet header and data fields. The socket listener's initialization module, `SL_InitSktListener`, shown in Listing 7-5 on page 7-24, releases the first element or buffer of the free queue and points to it from the current queue element (`current_qelem`) variable; it is this buffer that the socket listener uses when the .MPP driver calls the socket listener with a packet for it to process.

After the socket listener fills in the fields of the record pointed to by `current_qelem` with the processed contents of the packet, it moves the buffer into the used queue, pointed to by `used_queue`, for the client application to read. Then the socket listener releases the next record buffer from the free queue and points to it using the `current_qelem` variable. The sample code in Listing 7-7 on page 7-31 shows that when the client application has finished reading the contents of a used queue buffer element, it returns the buffer to the free queue pointed to by `free_queue` to make the buffer available again to the socket listener.

The socket listener uses the variables declared in Listing 7-1 to point to

- the free queue's queue header
- the used queue's queue header
- the current buffer queue element

Listing 7-1 Declarations for pointers to the sample socket listener's queues and packet buffer

```
SL_Locals    PROC
         ENTRY free_queue,used_queue,current_qelem
free_queue        DC.L    0      ;pointer to freeQ QHdr ;
                                 ; initialized by InitSktListener
used_queue        DC.L    0      ;pointer to usedQ QHdr ;
                                 ; initialized by InitSktListener
current_qelem     DC.L    0      ;pointer to current
                                 ; PacketBuffer record
ENDP;
```

Listing 7-4 on page 7-23 shows the Pascal-language client application `SetUpSocketListener` procedure. This procedure calls the `SL_InitSktListener` function to pass to the socket listener pointers to these two operating-system queues.

When the .MPP driver calls the socket listener, if there is an available buffer, the socket listener processes the packet and returns in the fields of the packet buffer record the DDP type, the destination node ID, the source address in `AddrBlock` format, the hop count, the size of the packet, a flag to indicate whether a checksum error occurred, and the data delivered in the packet. If you use the sample record data structure as a model, you can extend it to include fields to hold additional values, such as the tick count at the time when the .MPP driver called your socket listener. Listing 7-2 shows the assembly-language declaration for the `PacketBuffer` record.

Listing 7-2 Declaration for the sample socket listener's packet buffer record

```
PacketBuffer          RECORD    0
qLink                 DS.L      1
qType                 DS.W      1
buffer_Type           DS.W      1       ;DDP protocol type
buffer_NodeID         DS.W      1       ;destination node
buffer_Address        DS.L      1       ;source address in AddrBlock format
buffer_Hops           DS.W      1       ;hop count
buffer_ActCount       DS.W      1       ;length of DDP datagram
buffer_CheckSum       DS.W      1       ;chksum error returned here
                                        ; (cksumErr or noErr)
buffer_Data           DS.B      ddpMaxData
                                        ;the DDP datagram
                      ENDR
```

Listing 7-3 shows the socket listener's declaration for the queue header record, which is defined and used to make the code easier to read.

Listing 7-3 Declaration for the sample socket listener's queue header record

```
QHdr      RECORD    0
qFlags    DS.W      1
qHead     DS.L      1
qTail     DS.L      1
          ENDR
```

Setting Up the Socket Listener

The client application that includes the sample socket listener uses a Pascal procedure, `SetUpSocketListener`, to set up the socket listener's initialization routine. The `SetUpSocketListener` procedure defines

- the free and used queue variables of type `QHdr`
- a packet buffer record of type `PacketBuffer` to match the data structure defined in the socket listener code (The sample Pascal code declares an array of 10 packet buffer records.)

If you base your own code on the sample code, you can add new fields to the record declaration, if you need them. If you do this, you must modify the packet buffer data structure defined in the socket listener code to match the high-level language record declaration.

Listing 7-4 shows the client-application's Pascal code that initializes the packet buffer records and then adds them to the free queue using the `_Enqueue` trap. The code calls the `SL_InitSktListener` routine and passes to it pointers to the queue header for the free queue and the queue header for the used queue.

Listing 7-4 Setting up the socket listener from the client application

```
CONST
   ddpMaxData = 586;
TYPE
   PacketBuffer = RECORD
      qLink: QElemPtr;
      qType: Integer;
      buffer_Type: Integer;
      buffer_NodeID: Integer;
      buffer_Address: AddrBlock;
      buffer_Hops: Integer;
      buffer_ActCount: Integer;
      buffer_CheckSum: OSErr;
      buffer_Data: ARRAY[1..ddpMaxData] OF SignedByte;
END;

VAR
   freeQ, usedQ: QHdr;
   Buffers: ARRAY[1..10] OF PacketBuffer;

PROCEDURE SL_TheListener;
External;

FUNCTION SL_InitSktListener (freeQ, usedQ: QHdrPtr): OSErr;
External;

PROCEDURE SetUpSocketListener;
   VAR
      err: OSErr;
      i: Integer;
   BEGIN
      freeQ.QHead := NIL;     {initialize to nil to indicate empty queue}
      freeQ.QTail := NIL;     {initialize to nil to indicate end of queue}
      usedQ.QHead := NIL;     {initialize to nil to indicate empty queue}
      usedQ.QTail := NIL;     {initialize to nil to indicate end of queue}

      FOR i := 1 TO 10 DO     {add all buffers to the free queue}
          Enqueue(@Buffers[i], @freeQ);

      err := SL_InitSktListener(@freeQ, @usedQ);
                              {initialize the socket listener code}
```

```
      IF err <> noErr THEN
            BEGIN
               {Perform error processing here}
            END;
      {You can now call POpenSkt because the socket listener is ready to }
      { process packets.}
   END;
```

Initializing the Socket Listener

The sample socket-client application procedure `SetUpSocketListener` (shown in the preceding listing) calls the socket listener `SL_InitSktListener` initialization routine provided in Listing 7-5 to pass it pointers to the two operating-system queues (used and free) that the socket listener uses after the `SetUpSocketListener` procedure initializes these queues.

The `SL_InitSktListener` routine sets up its local variables `used_queue` and `free_queue` to point to the queue headers for the two queues. Then the routine releases from the free queue the first buffer and sets the `current_qelem` variable to point to it. This is the buffer that the socket listener uses when it next reads a packet.

Listing 7-5 Initializing the socket listener

```
;Function SL_InitSktListener(freeQ, usedQ: QHdrPtr): OSErr;
;
SL_InitSktListener PROC EXPORT

StackFrame     RECORD    {A6Link},DECR ;build a stack frame record
Result1        DS.W      1             ;function's result returned to caller
ParamBegin     EQU       *             ;start parameters after this point
freeQ          DS.L      1             ;freeQ parameter
usedQ          DS.L      1             ;usedQ parameter
ParamSize      EQU       ParamBegin-*  ;size of all the passed parameters
RetAddr        DS.L      1             ;placeholder for return address
A6Link         DS.L      1             ;placeholder for A6 link
LocalSize      EQU       *             ;size of all the local variables
               ENDR

   WITH        StackFrame,QHdr;        ;use these record types
   LINK        A6,#LocalSize           ;allocate your local stack frame

;Copy the queue header pointers into our local storage for use in the
; listener
```

```
   LEA         used_queue,A0           ;copy usedQ into used_queue
   MOVE.L      usedQ(A6),(A0)

   LEA         free_queue,A0           ;copy freeQ into free_queue
   MOVE.L      freeQ(A6),(A0)

;Release the first buffer record from freeQ and set current_qelem to it

   MOVEA.L     freeQ(A6),A1            ; A1 = ^freeQ
   LEA         current_qelem,A0        ;copy freeQ.qHead into current_qelem
   MOVE.L      qHead(A1),(A0)
   MOVEA.L     qHead(A1),A0            ;A0 = freeQ.qHead
   _Dequeue
   MOVE.W      D0,Result1(A6)          ;return status

@1 UNLK        A6                      ;destroy the link
   MOVEA.L     (SP)+,A0                ;pull off the return address
   ADDA.L      #ParamSize,SP           ;strip all of the caller's parameters
   JMP         (A0)                    ;return to the caller
   ENDP
   END
```

Processing a Packet

When the .MPP driver calls the sample socket listener, the socket listener's main module, the `SL_TheListener` procedure, reads and processes a packet addressed to the socket-client application. However, the socket listener can only process a packet if there is a packet buffer record available to hold the processed packet.

The code shown in Listing 7-6 determines if the `current_qelem` variable is `NIL` or not. If it is not `NIL`, the code gets a buffer, if one is available.

- If there is no buffer available, the code ignores the packet and calls the `ReadRest` routine with a buffer size value of 0. Before returning to the calling program, the code calls its `GetNextBuffer` routine to set up the `current_qelem` variable to point to the next available buffer, if there is one.
- If there is a buffer available, the code reads in the packet data and processes it.

If the socket listener reads the packet successfully, it processes the header information that the hardware driver has stored in the .MPP driver's local variable space pointed to by the value in register A2. To do this, the socket listener

- fills in a value for the hop count field of the packet buffer record and determines the packet length
- determines whether the DDP header is short or long and fills in the remaining fields of the packet buffer

- tests the checksum field of long DDP headers to determine if they are nonzero, indicating that the packet contains a checksum, and, if so, calculates the checksum
- adds the packet buffer to the used queue and then gets the next free buffer from the free queue and points to it with `current_qelem`

The socket listener then returns control to the calling program and waits until the .MPP driver calls it again when the .MPP driver next receives a packet addressed to a socket that is associated with the socket listener. Listing 7-6 shows the `SL_TheListener` procedure.

Listing 7-6 Receiving and processing a DDP packet

```
;SL_TheListener
;Input:
;     D0 (byte) = packet's destination socket number
;     D1 (word) = number of bytes left to read in packet
;     A0 points to the bytes to checksum
;     A1 points to the bytes to checksum
;     A2 points to MPP's local variables
;     A3 points to next free byte in read-header area
;     A4 points to ReadPacket and ReadRest jump table
;
;Return:
;     D0 is modified
;     D3 (word) = accumulated checksum

SL_TheListener  PROC    EXPORT
   WITH    PacketBuffer

;Get pointer to current PacketBuffer.
GetBuffer:
   LEA       current_qelem,A3    ;get the pointer to PacketBuffer
   MOVE.L    (A3),A3
   MOVE.L    A3,D0               ;if no PacketBuffer
   BEQ.S     NoBuffer            ; then ignore packet

;Read rest of packet into PacketBuffer.datagramData.
   MOVE.L    D1,D3               ;read rest of packet
   LEA       buffer_data(A3),A3  ;A3 = ^bufferData
   JSR       2(A4)               ;call ReadRest
   BEQ.S     ProcessPacket       ;if no error, continue
   BRA       RcvRTS              ;if error, ignore the packet
;No buffer; ignore the packet.
NoBuffer      CLR D3             ;set to ignore packet (buffer size = 0)
```

```
   JSR       2(A4)               ;call ReadRest
   BRA       GetNextBuffer       ;no buffer available, so read next packet;
                                 ; maybe there will be a buffer
                                 ; for the next packet

;Process the packet you just read in.
; ReadRest has been called so registers A0-A3 and D0-D3 are free
; to use. Use registers this way:
PktBuff         EQU    A0     ;current PacketBuffer
MPPLocals       EQU    A2     ;pointer to MPP's local variables
                              ; (still set up from entry to
                              ; socket listener)
HopCount        EQU    D0     ;gets the hop count
DatagramLength  EQU    D1     ;determines the datagram length
SourceNetAddr   EQU    D2     ;builds the source network address
ProcessPacket:
   LEA        current_qelem,PktBuff
                              ;PktBuff =  current_qelem
   MOVE.L     (PktBuff),PktBuff

;Do everything that's common to both long and short DDP headers
; first, clear buffer_Type and buffer_NodeID to ensure their high
; bytes are 0.
   CLR.W      buffer_Type(PktBuff)
                              ;clear buffer_Type
   CLR.W      buffer_NodeID(PktBuff)
                              ;clear buffer_NodeID

;Clear SourceNetAddr to prepare to build network address.
   MOVEQ      #0,SourceNetAddr   ;build the network address in
                              ; SourceNetAddr
;Get the hop count
   MOVE.W     toRHA+lapHdSz+ddpLength(MPPLocals),HopCount
                              ;get hop/length field
   ANDI.W     #DDPHopsMask,HopCount
                              ;mask off the hop count bits
   LSR.W      #2,HopCount        ;shift hop count into low bits
                              ; of high byte
   LSR.W      #8,HopCount        ;shift hop count into low byte
   MOVE.W     HopCount,buffer_Hops(PktBuff)
                              ; and move it into the
                              ; PacketBuffer
```

```
;Get the packet length (including the DDP header).
   MOVE.W     toRHA+lapHdSz+ddpLength(MPPLocals),DatagramLength
                                ;get length field
   ANDI.W     #ddpLenMask,DatagramLength
                                ;mask off the hop count bits

;Now, find out if the DDP header is long or short.
   MOVE.B     toRHA+lapType(MPPLocals),D3
                                ;get LAP type
   CMPI.B     #shortDDP,D3      ;is this a long or short DDP
                                ; header?
   BEQ.S      IsShortHdr        ;skip if short DDP header

;It's a long DDP header.
   MOVE.B toRHA+lapHdSz+ddpType(MPPLocals),buffer_Type+1(PktBuff)
                                ;get DDP type
   MOVE.B
      toRHA+lapHdSz+ddpDstNode(MPPLocals),buffer_NodeID+1(PktBuff)
                                ;get destination node from frame header
   MOVE.L     toRHA+lapHdSz+ddpSrcNet(MPPLocals),SourceNetAddr
                                ;source network in high word,
                                ; source node in low byte
   LSL.W      #8,SourceNetAddr  ;shift source node up to high byte
                                ; of low word; get source socket
                                ; from DDP header
   MOVE.B     toRHA+lapHdSz+ddpSrcSkt(MPPLocals),SourceNetAddr
   SUB.W      #ddpType+1,DatagramLength
                                ;DatagramLength = number of
                                ; bytes in datagram
   BRA.S      MoveToBuffer

;Determine if there is a checksum.
   TST.W      toRHA+lapHdSz+ddpChecksum(MPPLocals)
                                ;does packet have checksum?
   BEQ.S      noChecksum

;Calculate checksum for the DDP header.
   MOVE.L     DatagramLength,-(SP);save DatagramLength (D1)
   CLR        D3                ;set checksum to 0
   MOVEQ      #ddphSzLong-ddpDstNet,D1
                                ;D1 = length of header part to
                                ; checksum pointer to destination
                                ; network number in DDP header
```

```
   LEA        toRHA+lapHdSz+ddpDstNet(MPPLocals),A1
   JSR        SL_DoChksum         ;checksum of DDP header part
                                  ; (D3 holds accumulated
                                  ; checksum)

;Calculate checksum for the data portion of the packet (if any).
   MOVE.L     buffer_Data(PktBuff),A1
                                  ;pointer to datagram
   MOVE.L     (SP)+,DatagramLength
                                  ;restore DatagramLength (D1)
   MOVE.L     DatagramLength,-(SP)
                                  ;save DatagramLength (D1)
                                  ; before calling SL_DoChksum
   BEQ.S      TestChecksum        ;don't checksum datagram if
                                  ; its length = 0
   JSR        SL_DoChksum         ;checksum of DDP datagram part
                                  ; (D3 holds accumulated checksum)

TestChecksum:
   MOVE.L     (SP)+,DatagramLength
                                  ;restore DatagramLength (D1)

;Now make sure the checksum is OK.
   TST.W      D3                  ;is the calculated value 0?
   BNE.S      NotZero             ;if nonzero, go and use it
   SUBQ.W     #1,D3               ;if 0, make it -1

NotZero:
   CMP.W      toRHA+lapHdSz+ddpChecksum(MPPLocals),D3
   BNE.S      ChecksumErr         ;bad checksum
   MOVE.W     #0,buffer_CheckSum(A0)
                                  ;no errors
   BRA.S      noChecksum

ChecksumErr:
   MOVE.W     #ckSumErr,buffer_CheckSum(PktBuff)
                                  ;checksum error

noChecksum:
   BRA.S      MoveToBuffer
```

```
;It's a short DDP header.
IsShortHdr:
   MOVE.B     toRHA+lapHdSz+sddpType(MPPLocals),buffer_Type+1(PktBuff)
                                ;get DDP type
   MOVE.B     toRHA+lapDstAdr(MPPLocals),buffer_NodeID+1(PktBuff)
                                ;get destination node from LAP header
   MOVE.B     toRHA+lapSrcAdr(MPPLocals),SourceNetAddr
                                ;get source node from LAP header
   LSL.W      #8,SourceNetAddr   ;shift src node up to high byte of low word
   MOVE.B     toRHA+lapHdSz+sddpSrcSkt(MPPLocals),SourceNetAddr
                                ;get source socket from short DDP header
   SUB.W      #sddpType+1,DatagramLength
                                ;DatagramLength = number of bytes in
                                ; datagram
MoveToBuffer:
   MOVE.L     SourceNetAddr,buffer_Address(PktBuff)
                                ;move source network address into
                                ; PacketBufffer
   MOVE.W     DatagramLength,buffer_ActCount(PktBuff)
                                ;move datagram length into PacketBuffer

;Write the packet into the used queue and
; get another buffer from the free queue for the next packet.
   LEA        used_queue,A1      ;A1 = ^used_queue
   MOVE.L     (A1),A1            ;A1 = used_queue (pointer to usedQ)
   _Enqueue                      ;put the PacketBuffer in the used queue

GetNextBuffer:
   LEA        free_queue,A1      ;A1 = ^free_queue
   MOVE.L     (A1),A1            ;A1 = free_queue (pointer to freeQ)
   LEA        current_qelem,A0   ;copy freeQ.qHead into current_qelem
   MOVE.L     qHead(A1),(A0)
   MOVEA.L    qHead(A1),A0       ;A0 = freeQ.qHead
   _Dequeue

RcvRTS:
   RTS                           ;return to caller
   ENDP
```

Testing for Available Packets

Your client application must include a routine that determines if the socket listener has processed a packet for a socket associated with your client application. If it has, your client application routine must itself read and process the packet's contents, which are made available by the socket listener.

If your client application includes several processes each with its own socket that use the same socket listener, your client application routine must include a mechanism to scan for packets addressed to specific sockets.

If you expect to receive multiple packets for a specific socket, you should anticipate the possibility that the client application might handle the first packet for a socket before the socket listener processes the second packet for that socket. For example, to prepare for reception of multiple related packets addressed to the same socket, the sample client application's routine could check the socket listener's used queue `QHead` field for additional packets periodically after it read the first packet.

If you design your socket listener based on the sample one, your client's application should define a sufficient number of packet buffers so that as the client application releases a buffer from the used queue, processes its contents, and then moves that buffer back into the free queue for the socket listener to use, there are always buffers available in the free queue.

Listing 7-7 shows the code that the sample client application uses for this purpose. It periodically checks the `QHead` element of the socket listener's used queue. When `QHead` is not `NIL`, the client application knows that a packet is available for processing.

Listing 7-7 Determining if the socket listener has processed a packet

```
TYPE
   PacketBuffer = RECORD
      qLink: QElemPtr;
      qType: Integer;
      buffer_Type: Integer;
      buffer_NodeID: Integer;
      buffer_Address: AddrBlock;
      buffer_Hops: Integer;
      buffer_ActCount: Integer;
      buffer_CheckSum: OSErr;
      buffer_Data: ARRAY[1..ddpMaxData] OF SignedByte;
END;
PacketPtr = ^PacketBuffer;

VAR
   freeQ, usedQ: QHdr;
   bufPtr : PacketPtr;
```

```
         .
         .
         .
   WHILE (usedQ.QHead <> nil) DO
      BEGIN
         bufPtr := PacketPtr(usedQ.QHead); {get the packet ptr}
         IF (Dequeue(QElemPtr(bufPtr), @usedQ) <> noErr) THEN
            BEGIN
                                       {process the packet information}
            Enqueue(QElemPtr(bufPtr), @freeQ);
                                       {requeue the packet buffer for use}
      END
         ELSE
            BEGIN
               {Error occurred dequeueing packet - perform error }
               { processing here. However, because this is the only }
               { place in the code where buffers are dequeued, your error }
               { code should never be called. You can include a debugging }
               { statement here.}
   END;
END;
```

Measuring Packet-Delivery Performance

You use the AppleTalk Echo Protocol (AEP) to measure the performance of an AppleTalk network. Knowing the approximate speed at which an AppleTalk internet delivers packets is helpful in tuning the behavior of an application that uses one of the higher-level AppleTalk protocols, such as ATP and ADSP. You can also use AEP to test whether a node is on the network.

To tune an application, you need to know the round-trip time of a packet between two nodes on an AppleTalk internet. This is dependent on such factors as the network configuration, the number of routers and bridges that a packet must traverse, and the amount of traffic on the network; as these change, so does the packet transmission time. Routines belonging to the interfaces of both ATP and ADSP let you specify retry count and interval numbers whose optimum values you can better assess if you know the average round-trip time of a packet on your application's network.

AEP is implemented in each node as a DDP client process referred to as the ***AEP Echoer.*** The AEP Echoer uses a statically assigned socket, socket number 4, known as the ***echoer socket.*** The AEP Echoer listens for packets received through this socket.

Whenever it receives a packet, the AEP Echoer examines the packet's protocol type field to determine if the packet is an AEP packet, indicated by a value of 4. If it is, the first byte of the data portion of the packet serves as a function field. AEP uses two function codes:

- A value of 1 identifies the packet as an Echo Request packet.
- A value of 2 identifies the packet as an Echo Reply packet.

The AEP Echoer sets this field to a value of 2 to indicate that the packet is now a reply packet, then it calls DDP to send a copy of the packet back to the socket from which it originated. The AEP packet that you send is referred to as an ***Echo Request packet;*** the modified AEP packet that the AEP Echoer sends back to you is referred to as an ***Echo Reply packet.***

Here are some general guidelines that you should follow in using the AEP Echoer:

- Use the maximum packet size that you plan on using in your application.
- To test if a node is on the network, send several packets to that node because DDP can sometimes drop a packet.
- To test packet-delivery performance, send more than one packet and calculate the average round-trip time.

  Typically, you should receive an Echo Reply packet within a few milliseconds. If you do not get a response after about 10 seconds, you can assume that DDP dropped or lost your Echo Request packet, and you should resend the packet.

  The Echo Reply packet contains the same data that you sent in the Echo Request packet. If you send multiple packets to determine an average turnaround time and to compensate for the possibility of lost or dropped packets, you should include different data in the data portion of each packet; this will allow you to distinguish between replies to different request packets in the event that some replies are not delivered in the same order that you sent them or that some packets are dropped.
- To test packet-delivery performance time, your socket listener can include a field in its packet buffer record that saves the time in ticks when you sent the packet to compare against the response time.
- Accept only packets from the target node. Use your socket listener to filter out packets from nodes other than the target node to which you sent the Echo Request packet.

Follow these steps to send a packet to a target node and have AEP echo that packet back to your socket listener:

1. Write a socket listener to be used to receive an Echo Reply packet back from the target node to which you are sending the Echo Request packet.

   The AEP Echoer will send the Echo Reply packet to the socket from which you send the Echo Request packet. Follow the general instructions described earlier in this chapter that explain how to write a socket listener.
2. Call the `POpenSkt` function to open a socket from which to send an Echo Request packet, and assign your socket listener to that socket.

3. Determine the internet address of the target node to which you want to send an Echo Request packet.

   You can use the Name-Binding Protocol (NBP) to get the address of the destination application for which you want to measure round-trip packet delivery, and substitute the socket ID of the AEP Echoer; the socket number of the AEP Echoer is always 4 on every node. NBP routines are described in the chapter "Name-Binding Protocol (NBP)" in this book.

4. Prepare the datagram to be sent to the AEP Echoer on the target node by building a write-data structure with specific values for certain fields. You can use the `BuildDDPwds` procedure for this purpose.

   Set the destination socket number equal to 4 to indicate that it's the Echoer socket; set the DDP protocol type field also equal to 4 to indicate that the packet belongs to the AEP implementation on the target node; set the first byte of the data portion equal to 1 to indicate that this is an Echo Request packet. Fill in the destination network number and node ID for the target system; these are the numbers that NBP returned to you (see the preceding step).

5. Call the `PWriteDDP` function to send the Echo Request to the target node. As the value of `the wdsPointer parameter`, specify the pointer to the write data structure that you created.

DDP Reference

This section describes the data structures and routines that are specific to DDP. The "Data Structures" section shows the Pascal data structures for the records and parameter block that functions use for the protocol interface. The "Routines" section describes the DDP routines.

Data Structures

This section describes the data structures that you use to provide information to and receive it from DDP. It includes

- the write-data structure
- the address block record
- the MPP parameter block

The Write-Data Structure

A write-data structure is of type `WDSElement` and contains a series of pairs of length words and pointers. Each pair indicates the length and location of a portion of the data, including the header information, that constitutes the packet to be sent over the network.

You pass the `PWriteDDP` function a pointer to a write-data structure to send a DDP packet to another socket. You can use the `BuildDDPwds` procedure described on page 7-42 to create a write-data structure.

```
TYPE  WDSElement =
RECORD
   entryLength:   Integer;
   entryPtr:      Ptr;
END;
```

Field descriptions

`entryLength` The length of the data pointed to by `entryPtr`.

`entryPtr` A pointer to the DDP packet data to be sent using the `PWriteDDP` function.

The Address Block Record

The address block record defines a data structure of `AddrBlock` type. The `destAddress` parameter of the `BuildDDPwds` procedure takes an AppleTalk internet address value specified in this format.

You use NBP routines to get the address of an application that is registered with NBP. For more information about these routines, see the chapter "Name-Binding Protocol (NBP)" in this book.

```
TYPE AddrBlock =
PACKED RECORD
   aNet:          Integer;    {network number}
   aNode:         Byte;       {node ID}
   aSocket:       Byte;       {socket number}
END;
```

Field descriptions

`aNet` The number of the network to which the node belongs that is running the DDP client application whose address you are specifying.

`aNode` The node ID of the machine running the DDP client application whose address you are specifying.

`aSocket` The number of the socket used for the DDP client application.

MPP Parameter Block

The DDP `POpenSkt`, `PCloseSkt`, and `PWriteDDP` functions use the following variant record of the MPP parameter block, defined by the `MPPParamBlock` data type, to pass information to and receive it from the .MPP driver.

This section defines the fields that are common to all of the DDP functions that use the MPP parameter block. (The `BuildDDPwds` procedure does not use the MPP parameter block.) The fields that are used for specific functions only are defined in the descriptions of the functions to which they apply. This section does not define reserved fields, which are used either internally by the .MPP driver or not at all.

```
TYPE MPPParamBlock =
PACKED RECORD
   qLink:               QElemPtr;   {reserved}
   qType:               Integer;    {reserved}
   ioTrap:              Integer;    {reserved}
   ioCmdAddr:           Ptr;        {reserved}
   ioCompletion:        ProcPtr;    {completion routine}
   ioResult:            OSErr;      {result code}
   ioNamePtr:           StringPtr;  {reserved}
   ioVRefNum:           Integer;    {reserved}
   ioRefNum:            Integer;    {driver reference number}
   csCode:              Integer;    {primary command code}
   CASE MPPParmType OF
      OpenSktParm,
      CloseSktParm,
      WriteDDPParm:
      (
         socket:        Byte;       {socket number}
         checksumFlag:  Byte;       {checksum flag}
         listener:      Ptr;        {For POpenSkt, pointer to socket }
                                    { listener routine. For PWriteDDP, }
                                    { pointer to write-data structure.}
```

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute the `POpenSkt` function asynchronously, DDP calls your completion routine when it completes execution of the function. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute the `POpenSkt` function synchronously, it ignores the `ioCompletion` field.

`ioResult` The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.

`ioRefNum`	The .MPP driver reference number. The MPW interface fills in this field.
`csCode`	The command code of the MPP command to be executed. The MPW interface fills in this field.
`socket`	The number of the socket to be opened, closed, or from which to send data.

Routines

This section describes these DDP interface routines:

- the `POpenSkt` function that you use to open a DDP socket
- the `PCloseSkt` function that you use to close a socket that you opened with the `POpenSkt` function
- the `PWriteDDP` function that you use to send a datagram to another socket
- the `BuildDDPwds` procedure that you use to create a data structure to hold the header and data information that you want DDP to send

You pass parameters to and receive them from DDP in the fields of the parameter block whose pointer you pass directly to the routine that you call. An arrow preceding each parameter indicates whether it is an input parameter, an output parameter, or both:

Arrow	Meaning
→	Input
←	Output
↔	Both

Opening and Closing DDP Sockets

DDP delivers datagrams from socket to socket. You must open a socket before you use DDP to send or receive a DDP datagram.

- You use the `POpenSkt` function to open a DDP socket and associate your socket listener with it.
- You use the `PCloseSkt` function to close a socket that you opened with the `POpenSkt` function.

To receive a DDP datagram from another socket, you must provide a socket listener to receive packets and your own routine to read the data. When you open a socket, you specify a pointer to the socket listener for that socket.

POpenSkt

The `POpenSkt` function opens a socket for your application to use, and it adds that socket to the socket table along with a pointer to the socket listener that you provide.

```
FUNCTION POpenSkt (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether or not the function should be executed asynchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`csCode`	`Integer`	Always `openSkt` for this function.
↔	`socket`	`Byte`	The socket number.
→	`listener`	`Ptr`	A pointer to socket listener.

Field descriptions

`socket` The number of the socket you wish to open. Specify 0 for this field to have DDP assign a socket number in the range 128 through 254 and return it in this field. Socket numbers 1 through 63 are reserved for use by Apple Computer, Inc. You can use socket numbers 64 through 127 for this field during program development; however, it is recommended that you not use these numbers in a commercial product as there is no mechanism for resolving conflicts in the case that someone else uses the same socket number.

`listener` Pointer to a socket listener that you provide. You cannot specify `NIL` for this field. See "A Sample Socket Listener" beginning on page 7-20 for information on writing a socket listener.

DESCRIPTION

The `POpenSkt` function opens a DDP socket and associates that socket with the socket listener whose pointer you specify. If you specify 0 for the `socket` field, DDP dynamically assigns a socket, which it opens, and DDP returns the number of that socket to you.

Alternatively, you can specify a socket number as the value of the socket field. The `POpenSkt` function returns a result code of `ddpSktErr` if any of the following conditions is true:

- You specify the number of an already open socket.
- You pass a socket number greater than 127.
- The socket table is full.

The `POpenSkt` function is equivalent to calling the `PBControl` function with a value of `openSkt` in the `csCode` field of the parameter block.

You must provide a socket listener when you call the `POpenSkt` function. If you do not intend to listen for DDP datagrams through the socket you open with this function, you can provide a socket listener that does nothing but immediately return control to DDP.

DDP reads the destination socket address and delivers datagrams to the socket listener associated with the socket. The socket listener can be part of a DDP client application or a higher-level AppleTalk protocol that is also a client of DDP.

If you want a process using a socket to be visible to other processes using the AppleTalk network, use the NBP `PRegisterName` function to register the name that is associated with the socket and address of the process. See the chapter "Name-Binding Protocol (NBP)" in this book for more information about NBP.

SPECIAL CONSIDERATIONS

You cannot specify `NIL` for the `listener` parameter; if you do so, your application will crash and the computer on which it is running will hang.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `POpenSkt` function from assembly language, call the `_Control` trap macro with a value of `openSkt` in the `csCode` field of the parameter block. You must also specify the .MPP driver reference number. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`ddpSktErr`	–91	Bad socket number or socket table is full

SEE ALSO

For information about how to use the `POpenSkt` function in sequence with other routines to send and receive data over an AppleTalk network, see "Sending and Receiving Data: An Overview" beginning on page 7-9.

PCloseSkt

The `PCloseSkt` function removes the entry for a specific socket from the socket table.

```
FUNCTION PCloseSkt (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr`	A pointer to an MPP parameter block.
`async`	A Boolean that specifies whether or not the function should be executed asynchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A completion routine.
←	ioResult	OSErr	The result code.
→	csCode	Integer	Always closeSkt for this function.
→	socket	Byte	The number of the socket to close.

Field descriptions

`socket` The number of the socket you wish to close. You cannot use 0 for this field.

DESCRIPTION

Use the `PCloseSkt` function to close a socket that you opened with the `POpenSkt` function. The `PCloseSkt` function returns a result code of `ddpSktErr` if you specify a socket number of 0 or if there is no open socket with the socket number you specify.

The `PCloseSkt` function is equivalent to calling the `PBControl` function with a value of `closeSkt` in the `csCode` field of the parameter block.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PCloseSkt` function from assembly language, call the `_Control` trap macro with a value of `closeSkt` in the `csCode` field of the parameter block. You must also specify the .MPP driver reference number. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`ddpSktErr`	–91	Bad socket number

SEE ALSO

For information on the assignment of socket numbers, see "POpenSkt" beginning on page 7-38.

Sending DDP Datagrams

To send a DDP datagram to another socket, you must first open a socket with the `POpenSkt` function, prepare a write-data structure, and finally send the packet using the `PWriteDDP` function described in this section. You can use the `BuildDDPwds` procedure described in this section to create the write-data structure.

PWriteDDP

The `PWriteDDP` function sends a DDP datagram to another socket.

```
FUNCTION PWriteDDP (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to an MPP parameter block.

`async` A Boolean that specifies whether the function should be executed asynchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`csCode`	`Integer`	Always `writeDDP` for this function.
→	`socket`	`Byte`	The number of socket to send data from.
→	`checksumFlag`	`Byte`	The checksum flag; nonzero to compute checksum.
→	`wdsPointer`	`Ptr`	A pointer to write-data structure.

Field descriptions

`socket` The number of the socket from which you want to send data. See the description of the `POpenSkt` function for information on the assignment of socket numbers.

`checksumFlag` The checksum flag. If you set this field to a nonzero value and if DDP uses a long header for the datagram (that is, if the destination socket has a network number different from that of the source socket), then the `PWriteDDP` function calculates a checksum for the datagram and includes it in the datagram header. Set this field to 0 if you do not want the `PWriteDDP` function to calculate a checksum.

`wdsPointer` A pointer to a write-data structure. The write-data structure provides the destination address and the data for the datagram. The DDP write-data structure is described in "Creating a DDP Write-Data Structure" on page 7-12.

DESCRIPTION

Before you call the `PWriteDDP` function, you must prepare a write-data structure. The write-data structure, shown in Figure 7-4 on page 7-13, includes a pointer to the destination address and pointers to buffers containing the data you wish to send. You can use the `BuildDDPwds` procedure to build a write-data structure.

Set the checksum flag field when you call the `PWriteDDP` function to have the function calculate the checksum and include it in the packet header. Note, however, that only long packet headers include a checksum field, and that whether the checksum is used for error checking depends on how the socket listener code at the destination socket is implemented.

The `PWriteDDP` function is equivalent to calling the `PBControl` function with a value of `writeDDP` in the `csCode` field of the parameter block.

SPECIAL CONSIDERATIONS

Memory used for the write-data structure belongs to DDP and must be nonrelocatable until the `PWriteDDP` function completes execution, after which you can either reuse the memory or release it.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PWriteDDP` function from assembly language, call the `_Control` trap macro with a value of `writeDDP` in the `csCode` field of the parameter block. You must also specify the .MPP driver reference number. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`ddpSktErr`	–91	Bad socket number
`ddpLenErr`	–92	Datagram data exceeds 586 bytes
`noBridgeErr`	–93	Could not find router to forward packet

SEE ALSO

For a description of the DDP write-data structure, see "Creating a DDP Write-Data Structure" on page 7-12.

If you are programming in Pascal or C, see the description of the `BuildDDPwds` procedure that follows for help in creating a write-data structure.

BuildDDPwds

The `BuildDDPwds` procedure creates a write-data structure that you can use to send a DDP packet to a remote socket.

```
PROCEDURE BuildDDPwds (wdsPtr,headerPtr,dataPtr: Ptr;
                       destAddress: AddrBlock; DDPType: Integer;
                       dataLen: Integer);
```

`wdsPtr` A pointer to a buffer that you provide that will contain the write-data structure. The write-data structure created by `BuildDDPwds` is 14 bytes long.

`headerPtr` A pointer to a buffer that you provide that will contain the packet header. This buffer must be at least 17 bytes long.

`dataPtr` A pointer to the data that you want to send. The maximum amount of data that you can include in a DDP data packet is 586 bytes.

`destAddress`
: The address of the socket to which you want to send the data. The address consists of the network number, the node ID, and the socket number in `AddrBlock` format; see "The Address Block Record" on page 7-35.

  A node ID of 255 is the broadcast address; that is, the datagram is broadcast to all nodes in the network. Note, however, that broadcast datagrams are not forwarded by routers and so are not sent to nodes on other networks in the internet.

`DDPType`
: The DDP protocol type of the packet you are sending. DDP protocol types 1 through 15 are reserved for use by Apple Computer, Inc. You can use other protocol types as you see fit.

`dataLen`
: The length of the data pointed to by the `dataPtr` parameter.

DESCRIPTION

The `BuildDDPwds` procedure creates a write-data structure that consists of a pointer for the header, a length word and pointer for the data, and a terminating 0 word. Because the first pointer in the write-data structure must point to an odd address, it is difficult to use Pascal to create a write-data structure. In this case, using the `BuildDDPwds` procedure simplifies the process. However, the `BuildDDPwds` procedure assumes that the data that you are sending is in a single block. In most cases, if you are using DDP directly to send data across a network, a single block of data should be adequate.

You must provide a 17-byte buffer for the header block, a 14-byte buffer to hold the write-data structure, and a pointer to the data you want to send. (The header block is only 16 bytes, but because it begins on an odd address, the first byte is not used.)

SPECIAL CONSIDERATIONS

Memory that you allocate for the write-data structure buffers belongs to DDP and must be nonrelocatable until the `PWriteDDP` function completes execution, after which you can either reuse the memory or release it.

ASSEMBLY-LANGUAGE INFORMATION

The `BuildDDPwds` procedure is implemented entirely in the MPW interface files. There is no assembly-language equivalent to this procedure.

SEE ALSO

The write-data structure is defined in "Creating a DDP Write-Data Structure" on page 7-12.

To send the data pointed to by your write-data structure, use the `PWriteDDP` function described on page 7-41.

Summary of DDP

Pascal Summary

Constants

```
CONST
   {.MPP driver unit and reference numbers}
   mppUnitNum     =        9;                {MPP unit number}
   mppRefNum      =      -10;                {MPP reference number}

   {csCodes}
   writeDDP       =      246;                {write out DDP packet, csCode}
   closeSkt       =      247;                {close DDP socket, csCode}
   openSkt        =      248;                {open DDP socket, csCode}
```

Data Types

The Write-Data Structure

```
TYPE  WDSElement =
   RECORD
      entryLength:   Integer;
      entryPtr:      Ptr;
   END;
```

The Address Block Record

```
TYPE AddrBlock =
   PACKED RECORD
      aNet:          Integer;    {network number}
      aNode:         Byte;       {node ID}
      aSocket:       Byte;       {socket number}
   END;
```

MPP Parameter Block

```
MPPParmType = (...OpenSktParm,CloseSktParm,WriteDDPParm ...)

TYPE MPPParamBlock =
   PACKED RECORD
      qLink:               QElemPtr;   {reserved}
      qType:               Integer;    {reserved}
      ioTrap:              Integer;    {reserved}
      ioCmdAddr:           Ptr;        {reserved}
      ioCompletion:        ProcPtr;    {completion routine}
      ioResult:            OSErr;      {result code}
      ioNamePtr:           StringPtr;  {reserved}
      ioVRefNum:           Integer;    {reserved}
      ioRefNum:            Integer;    {driver reference number}
      csCode:              Integer;    {command code}
      CASE MPPParmType OF
         OpenSktParm,
         CloseSktParm,
         WriteDDPParm:
         (
            socket:        Byte;       {socket number}
            checksumFlag:  Byte;       {checksum flag}
            listener:      Ptr;        {For POpenSkt, pointer to socket }
                                       { listener routine. For PWriteDDP, }
                                       { pointer to write-data structure.}
         )
   END;

MPPPBPtr = ^MPPParamBlock;
```

Routines

Opening and Closing DDP Sockets

```
FUNCTION POpenSkt          (thePBptr: MPPPBPtr; async: Boolean): OSErr;
FUNCTION PCloseSkt         (thePBptr: MPPPBPtr; async: Boolean): OSErr;
```

Sending DDP Datagrams

```
FUNCTION PWriteDDP         (thePBptr: MPPPBPtr; async: Boolean): OSErr;
PROCEDURE BuildDDPwds      (wdsPtr,headerPtr,dataPtr: Ptr;
                            destAddress: AddrBlock; DDPType: Integer;
                            dataLen: Integer);
```

C Summary

Constants

```
/*DDP parameter constants*/
#define MPPioCompletion MPP.ioCompletion
#define MPPioResult MPP.ioResult
#define MPPioRefNum MPP.ioRefNum
#define MPPcsCode MPP.csCode
#define DDPsocket DDP.socket
#define DDPchecksumFlag DDP.checksumFlag
#define DDPwdsPointer DDP.DDPptrs.wdsPointer
#define DDPlistener DDP.DDPptrs.listener

/*.MPP driver unit and reference number*/
enum {
   mppUnitNum     =        9,          /*MPP unit number*/
   mppRefNum      =      -10           /*MPP reference number*/
};

/*DDP csCodes*/
enum {
   writeDDP       =        246,        /*send DDP packet*/
   closeSkt       =        247,        /*close DDP socket*/
   openSkt        =        248         /*open DDP socket*/
};
```

Data Types

The Write-Data Structure

```
struct   WDSElement {
   short    entryLength;
   Ptr      entryPtr;
} WDSElement;
```

The Address Block Record

```
struct AddrBlock {
      short               aNet;               /*network number*/
      unsigned char       aNode;              /*node ID*/
      unsigned char       aSocket;            /*socket number*/
};

typedef struct AddrBlock AddrBlock;
```

MPP Parameter Block

```
#define MPPATPHeader\
   QElem          *qLink;        /*reserved*/\
   short          qType;         /*reserved*/\
   short          ioTrap;        /*reserved */\
   Ptr            ioCmdAddr;     /*reserved*/\
   ProcPtr        ioCompletion;  /*completion routine*/\
   OSErr          ioResult;      /*result code*/\
   long           ioNameptr;     /*command result (ATP user bytes)*/\
   short          ioVRefNum;     /*request transaction ID*/\
   short          ioRefNum;      /*driver reference number*/\
   short          csCode;        /*command code*/

typedef struct {
   MPPATPHeader
}MPPparms;

union ParamBlockRec {
   MPPparms    MPP;              /*general MPP parms*/
   DDPparms    DDP;              /*DDP calls*/
};
typedef MPPParamBlock   *MPPPBtr;

typedef struct {
MPPATPHeader
   char socket;                  /*socket number*/
   char checksumFlag;            /*checksum flag*/
   union {
   Ptr wdsPointer;               /*pointer to write-data structure*/
   Ptr listener;                 /*pointer to write-data structure or */
                                 /* pointer to socket listener*/
} DDPptrs;
}DDPparms;
```

Routines

Opening and Closing DDP Sockets

```
pascal OSErr POpenSkt       (MPPPBPtr the PBptr, Boolean async);
pascal OSErr PCloseSkt      (MPPPBPtr thePBptr, Boolean async);
```

Sending DDP Datagrams

```
pascal OSErr PWriteDDP      (MPPPBPtr the PBptr, Boolean async);
pascal void BuildDDPwds     (Ptr wdsPtr, header Ptr, Ptr dataPtr,
                             const AddrBlock netAddr, short ddpType,
                             short dataLen);
```

Assembly-Language Summary

Constants

```
mppUnitNum      EQU      9         ;MPP unit number

;csCodes for DDP
writeDDP        EQU      246       ;write out DDP packet
closeSkt        EQU      247       ;close DDP socket
openSkt         EQU      248       ;open DDP socket

;long DDP packet header
ddpHopCnt       EQU      0         ;hop count (byte)
ddpLength       EQU      0         ;packet length (word)
ddpChecksum     EQU      2         ;checksum (word)
ddpDstNet       EQU      4         ;destination network number (word)
ddpSrcNet       EQU      6         ;source network number (word)
ddpDstNode      EQU      8         ;destination node address (byte)
ddpSrcNode      EQU      9         ;source node address (byte)
ddpDstSkt       EQU      10        ;destination socket number (byte)
ddpSrcSkt       EQU      11        ;source socket number (byte)
ddpType         EQU      12        ;DDP protocol type field (byte)

;short DDP packet header
sddpDstSkt      EQU      2         ;destination socket number (byte)
sddpSrcSkt      EQU      3         ;source socket number (byte)
sddpType        EQU      4         ;DDP protocol type field (byte)

;DDP long header size
ddphSzLong      EQU      13        ;size of extended DDP header

DDP short header size
ddphSzShort     EQU      5         ;size of short DDP header

shortDDP        EQU      $01       ;LAP type code for DDP (short header)
longDDP         EQU      $02       ;LAP type code for DDP (long header)
```

```
;DDP miscellaneous
ddpMaxWKS       EQU     $7F     ;highest valid well-known socket
ddpMaxData      EQU     586     ;maximum DDP data size
ddpLenMask      EQU     $03FF   ;mask for DDP length
rhaSize         EQU     $18     ;size of read-header area
toRHA           EQU     1       ;top of the read-header area

wdsEntrySz      EQU     6       ;size of a write-data structure entry
DDPHopsMask     EQU     $3C00   ;mask hop count bits from field in DDP
                                ; header

;command codes (csCodes)
writeDDP        EQU     246     ;write out DDP packet
closeSkt        EQU     247     ;close DDP socket
openSkt         EQU     248     ;open DDP socket
```

Data Structures

MPP Parameter Block Common Fields for DDP Routines

0	qLink	long	reserved
4	qType	word	reserved
6	ioTrap	word	reserved
8	ioCmdAddr	long	reserved
12	ioCompletion	long	address of completion routine
16	ioResult	word	result code
18	ioNamePtr	long	reserved
22	ioVRefNum	word	reserved
24	ioRefNum	word	driver reference number

OpenSkt Parameter Variant

26	csCode	word	command code; always `openSkt`
28	socket	byte	socket number
30	listener	long	pointer to socket listener

CloseSkt Parameter Variant

26	csCode	word	command code; always `closeSkt`
28	socket	byte	number of socket to be closed

WriteDDP Parameter Variant

26	csCode	word	command code; always `writeDDP`
28	socket	byte	number of socket to write from
30	listener	long	pointer to write-data structure

Result Codes

`noErr`	0	No error
`ddpSktErr`	–91	Bad socket number or socket table is full
`ddpLenErr`	–92	Datagram data exceeds 586 bytes
`noBridgeErr`	–93	Could not find router to forward packet