This chapter describes how to write data directly to an Ethernet, token ring, or Fiber Distributed Data Interface (FDDI) driver. For Ethernet Phase 1 packets, that is, the original version of Ethernet packets, this chapter also describes how to read data directly from an Ethernet driver using either the default protocol handler that Apple provides or your own protocol handler.

For Phase 2 packets, that is, IEEE 802.2 packets, you must use the interface to the Link-Access Protocol (LAP) Manager to attach your protocol handler to read data from an Ethernet, token ring, or FDDI driver.

For a description of how to attach a protocol handler to read 802.2 packets, see the chapter "Link-Access Protocol (LAP) Manager" in this book, which also explains Ethernet Phase 1 packets and Phase 2 packets for Ethernet, token ring, and FDDI.

For an introduction to the hardware and software of an entire AppleTalk network, see *Understanding Computer Networks* and the *AppleTalk Network System Overview*. For information on designing circuit cards and device drivers for Macintosh computers, see *Designing Cards and Drivers for the Macintosh Family*, second edition.

To use this chapter, you should be familiar with the information on Ethernet and token ring provided in *Inside AppleTalk*, second edition. (*Inside AppleTalk* does not address FDDI.) To gain an understanding of the relationship between the AppleTalk data links and the physical device drivers, see the chapter "Introduction to AppleTalk" in this book, which also introduces some of the terminology used in this chapter.

About Ethernet, Token Ring, and FDDI Support

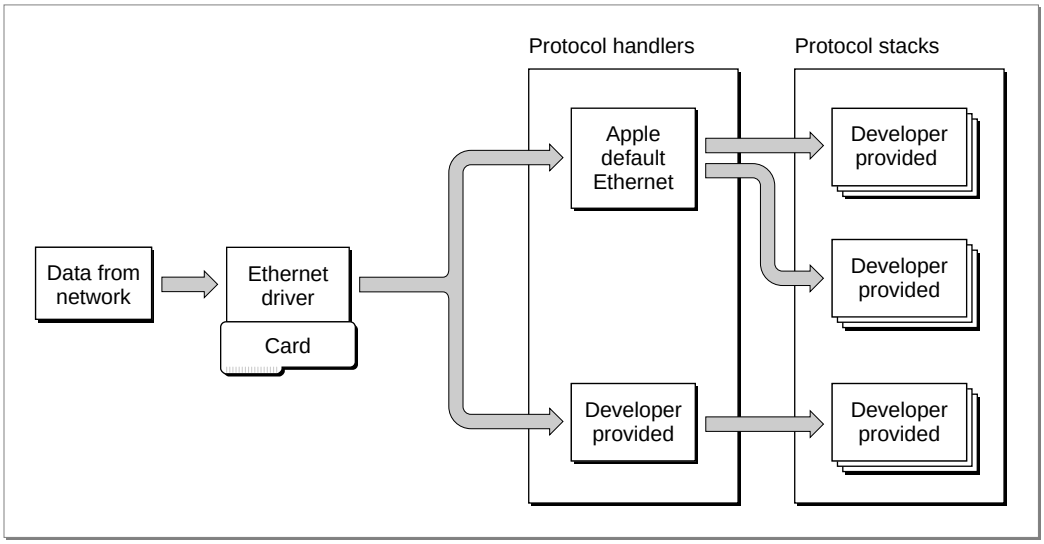
You can write an application that processes packets for a protocol other than AppleTalk and run your application on Macintosh computers that also run the AppleTalk protocol stack. To send data from your application, you need to communicate directly with a network hardware device driver. To read data, you either use the LAP Manager or directly communicate with the hardware device driver, depending on the type of packet that your application processes. To read data from the network hardware device driver, you must use a protocol handler, which is code that the driver calls, in this case, to process an incoming packet for a specific protocol type.

Ethernet Phase 1 packets are IEEE 802.3 protocol packets. If your application processes Ethernet Phase 1 packets, you can use the default protocol handler that Apple Computer, Inc. provides to read data addressed to the protocol type that your application handles, or you can create and attach your own protocol handler to read that data. The chapter "Link-Access Protocol (LAP) Manager" in this book provides more information about Phase 1 and Phase 2 packets, including figures that show the two packet formats.

For Ethernet Phase 1 packets, the Apple Ethernet implementation supports multiple protocol types, and more than one protocol handler can be attached to the Ethernet driver at the same time. For example, you can write an application implementing a protocol stack that uses the default Apple Ethernet protocol handler. Another developer can write an application implementing a different protocol stack, and it, too, can use the

default Apple Ethernet protocol handler. A third developer can write an application implementing yet another protocol stack that supplies and attaches its own protocol handler to the Ethernet driver. All of these applications can run concurrently on the same machine. Figure 11-1 shows three developer-provided applications that implement protocol stacks, all using the Ethernet driver at the same time.

Figure 11-1 Using protocol handlers to read data directly from the Ethernet driver



The Ethernet driver maintains a list that identifies the protocol handler for each protocol type. When you attach your protocol handler to the Ethernet driver, it adds an entry to its list for the type of protocol that your application supports along with the pointer to your protocol handler. When a packet arrives for your application, the driver reads the protocol type, locates the pointer to the protocol handler, and calls the protocol handler to read the data.

For all 802.2 packets, which includes Ethernet Phase 2 packets and all token ring and FDDI packets, Apple Computer, Inc. recommends that you attach your protocol handler using the LAP Manager interface. All AppleTalk packets are 802.2 packets. (For more information about using the LAP Manager to attach protocol handlers, see the chapter "Link-Access Protocol [LAP] Manager" in this book.)

At the hardware device driver level, only one protocol handler can be attached to receive 802.2 packets. Although you can attach more than one protocol handler at this level, if you do so, you will cause problems for AppleTalk. The AppleTalk protocol stack uses the LAP Manager's protocol handler for 802.2 packets to connect to a hardware device driver. (All AppleTalk packets are 802.2 packets.) If you attach your own protocol handler for 802.2 packets, the LAP Manager will be unable to attach its protocol handler, and you will have excluded AppleTalk from using the hardware device driver simultaneously.

For example, suppose a user is running your application with its own protocol handler over token ring and AppleTalk over Ethernet. If the user decides to change the AppleTalk network type to token ring, the attempted connection switch will fail because the LAP Manager will not be able to attach its protocol handler to the token ring device driver. To avoid problems such as these, Apple recommends that you attach your protocol handler to read Ethernet Phase 2, token ring, or FDDI 802.2 packets through the LAP Manager.

The LAP Manager installs a protocol handler at the hardware device driver level that receives 802.2 packets and that also serves as a dispatcher. This protocol handler maintains an index of registered protocol types and pointers to their protocol handlers, which allows the LAP Manager to act as a dispatcher, thereby permitting the concurrent use of a token ring or FDDI hardware device driver by more than one application, including AppleTalk.

Notes for applications that handle token ring and FDDI 802.2 packets

Apple provides specifications for both token ring and FDDI drivers that result in these implications for network applications:

- Only one protocol handler can be attached at the hardware device driver level.
- Only one protocol type is supported: the IEEE 802.2 Type 1 protocol that provides for a connectionless, or datagram, service.
- Apple does not provide a default protocol handler for token ring or FDDI.

These limitations do not restrict you from attaching your own protocol handler directly to a token ring or an FDDI hardware device driver, but doing so results in the consequences stated previously. ◆

About Multivendor Network Interface Controller (NIC) Support

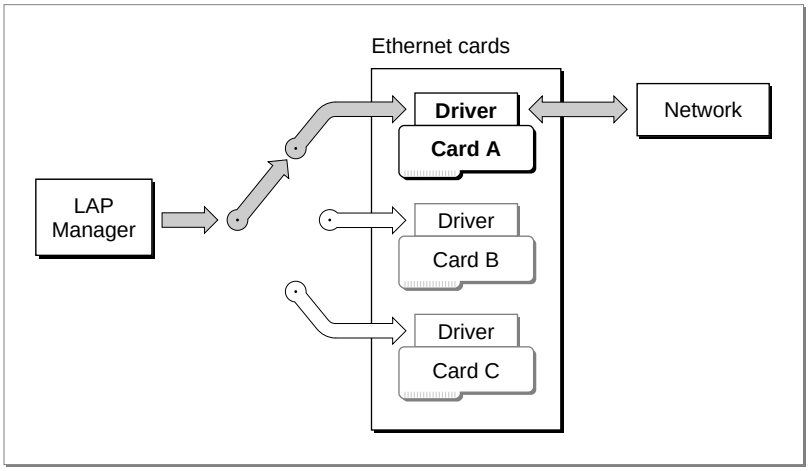
Before AppleTalk version 56, a networked Macintosh computer could support only one Ethernet or token ring connection at a time. This posed a limitation for many developers who wanted multiple concurrent Ethernet or token ring connections. The original architecture also lacked support for the concurrent use of a NuBus slot device and a non-NuBus device, such as a SCSI Ethernet connection or the processor-direct slot (PDS) device.

To solve this problem, Apple implemented ***multivendor architecture*** to provide support that allows you to use different brands of Ethernet, token ring, or FDDI NuBus hardware in the same machine at the same time. For example, multivendor architecture allows a single machine to run AppleTalk over one Ethernet card (or through an Ethernet network connector that uses the SCSI port) and to run another application that implements a different protocol, such as TCP/IP, over another Ethernet card at the same time.

The user can select the network type to be used depending on the NuBus cards and slotless devices that are installed in the Macintosh computer. In addition to supporting various types of network hardware, multivendor architecture allows AppleTalk users to also select which brand of card to use. Your application can also provide support that allows a user to select a particular brand of card for a particular type of network connection.

Figure 11-2 shows three different brands of Ethernet cards installed in a single machine and indicates the path that data follows from the LAP Manager through the driver of the selected Ethernet card and out to the network when AppleTalk is used. The user can choose which Ethernet card is used as the network connector.

Figure 11-2 How AppleTalk uses multivendor support



To make possible the use of multiple brands of network cards, Apple provides a driver shell for each of the following types of networks:

- For Ethernet, the name of the driver shell is .ENET.
- For token ring, the name of the driver shell is .TOKN.
- For FDDI, the name of the driver shell is .FDDI.

These driver shells consist of commands that locate and load the driver software for a particular card of that network type.

Note
For configurations that are not NuBus based, such as PDS-based and SCSI-based hardware solutions, you must open and use the following drivers, not the driver shells: .ENET0 for Ethernet, .TOKN0 for token ring, .FDDI0 for FDDI. ◆

The `EGetInfo` function returns information about the .ENET driver. If the Ethernet card that you are using has a SONIC chip, you can use the `EGetInfo` function to obtain information pertaining to the SONIC-based network interface controller (NIC). Beginning with version 58 of AppleTalk, the `EGetInfo` function returns this additional information. For the details regarding Ethernet cards with SONIC chips, see "EGetInfo" beginning on page 11-36.

About Multicast Addressing

At the hardware device driver level, Apple supports multicast addressing. A ***multicast address*** is a hardware address that is shared by a subset of nodes on a particular data link. This is similar in concept to a broadcast hardware address, but a multicast address is used to send directed broadcasts to the subset group of nodes only, and not to all nodes on the data link. A broadcast address is shared by all nodes on a particular type of network. Packets sent to the Ethernet broadcast address are sent to all nodes on the Ethernet data link. Ethernet and FDDI networks use multicast addresses; the token ring equivalent of a multicast address is a ***functional address.***

A network type, such as an Ethernet data link, can also have associated with it one or more multicast addresses. Each data link is identified by a unique hardware address to which packets for that network hardware are sent. In addition to this unique hardware address, a data link can receive packets that contain the broadcast address for its own network type—Ethernet, for example.

When a node on a data link transmits a packet that has a multicast hardware address as its destination hardware address, then only a specific subset of the nodes on the link will receive the packet. Each node can have any number of multicast addresses, and any number of nodes can have the same multicast address. Some nodes on the link may not have a multicast address; other nodes may have more than one multicast addresses. (For more information on multicast and functional addresses, see *Inside AppleTalk,* second edition. See also “EAddMulti” on page 11-40.)

Using Ethernet, Token Ring, and FDDI Drivers

This section describes how to write an application that implements a protocol other than AppleTalk and that reads data from and writes it to the hardware device driver for a particular network interface controller.

For Ethernet, this section describes how to locate the installed Ethernet cards and open the Ethernet driver for a particular card or a slotless device. Then it describes how to write data to the driver. Next it describes how to attach either the Apple default protocol handler or your own protocol handler to the Ethernet driver to read data for Ethernet Phase 1 packets.

For token ring and FDDI, this section describes the differences between using the Ethernet driver and the token ring or FDDI driver, including the steps to follow to read data from and write it to this driver.

Using the Ethernet Driver

You can write your own protocol stack or application that uses the Ethernet driver directly rather than going through the LAP Manager. Apple provides an .ENET driver shell that locates and loads the driver for the selected Ethernet NuBus card. The driver

shell searches the following locations for existing Ethernet driver resources, and it uses the most current one:

- the system resource file
- the card's declaration ROM
- the motherboard's ROM

See *Designing Cards and Drivers for the Macintosh Family,* second edition, for discussions of NuBus board IDs and slot resources.

Opening the Ethernet Driver

Before you use the Device Manager's `OpenSlot` function to open the .ENET driver, you use the `SGetTypeSRsrc` function described in the Slot Manager chapter of *Inside Macintosh: Devices* to determine which NuBus slots contain Ethernet cards. To find Ethernet NuBus cards, use the value `catNetwork` in the field `spCategory` of the `GetTypeSRsrc` function parameter block, and use the value `typeEthernet` in the field `spCType`. If you cannot find any Ethernet NuBus cards, you should also attempt to open the .ENET0 driver in case non-NuBus Ethernet hardware is attached to the system. You should provide a user interface that allows the user to select a specific Ethernet card in the case that more than one is present. (The chapter "Device Manager" in *Inside Macintosh: Devices* describes the `OpenSlot` function.)

Note
This section refers to the .ENET driver shell, which facilitates multivendor support, as the .ENET driver. When you open the .ENET driver shell, it loads and opens the particular card's driver. ◆

Listing 11-1 illustrates how to identify and open an Ethernet driver.

Listing 11-1 Finding an Ethernet card and opening the .ENET driver

```
FUNCTION Get_And_Open_ENET_Driver: Integer;
   VAR
      mySBlk:     SpBlock;
      myPBRec:    ParamBlockRec;
      myErr:      OSErr;
      Found:      Integer;
      ENETRefNum: Integer;
      EnetStr:    Str15;
      Enet0Str:   Str15;

BEGIN
   Found := 0;                  {assume no sResource found}
   ENETRefNum := 0;             {indicate no driver found}
```

```
   WITH mySBlk DO                {set up the SpBlock}
      BEGIN
         spParamData := 1;       {include search of disabled resources }
                                 { starting searching from spSlot and }
                                 { the slots above it}
         spCategory := catNetwork;
         spCType := typeEthernet;
         spDrvrSW := 0;
         spDrvrHW := 0;
         spTBMask := 3;          {match only Category and }
                                 { CType fields}
         spSlot := 0;            {start search from here}
         spID := 0;              {start search from here}
         spExtDev := 0;          {ID of the external device}
      END;
         .
         .
      {REPEAT}
         .
      {At this point you could implement a repeat loop to check }
      { for multiple Ethernet cards. This sample uses the first card.}
         .
myErr := SGetTypeSRsrc(@mySBlk);
IF myErr = noErr THEN            {found an sResource match; }
                                 { save it for later}
   BEGIN
      Found := Found + 1;
      (SaveSInfo(@mySBlk);       {save slot info for later use}
   END;
      {until myErr = smNoMoresRsrcs;}

 IF Found > 1 THEN
   BEGIN
      {If you find more than one sResource, put up a dialog box }
      { to let the user select one. If any of the sResources }
      { that you found were disabled, let the user know that they }
      { are not available.}
      {This code sample assumes that the selected slot is }
      { returned in mySBlk.spSlot, that the corresponding }
      { sResource ID is returned in mySBlk.spID, and that Found }
      { remains > 1 to indicate that it is okay to open the }
      { driver.}
   END;
```

```
   IF found <> 0 THEN
      BEGIN
         EnetStr := '.ENET';
         WITH myPBRec DO
            BEGIN
               ioCompletion := NIL;          {call made synchronously}
               ioNamePtr := @EnetStr;
               ioPermssn := fsCurPerm;
               ioFlags := 0;                 {reserved for driver use}
               ioSlot := mySBlk.spSlot;      {slot of Ethernet card to open}
               ioID := mySBlk.spID;          {sResource ID for slot}
            END;
         myErr := OpenSlot(@myPBRec, FALSE);
         IF myErr = noErr THEN
         ENETRefNum := myPBRec.ioRefNum;
      END
   ELSE
      BEGIN
         Enet0Str := '.ENET0';
         myErr := OpenDriver(Enet0Str, ENETRefNum);
      END;
   IF myErr <> noErr THEN
      DoError(myErr); {handle the error}
   Get_And_Open_ENET_Driver := ENETRefNum;   {return the refNum or }
                                             { 0 if unsuccessful}
END;
```

Using a Write-Data Structure to Transmit Ethernet Data

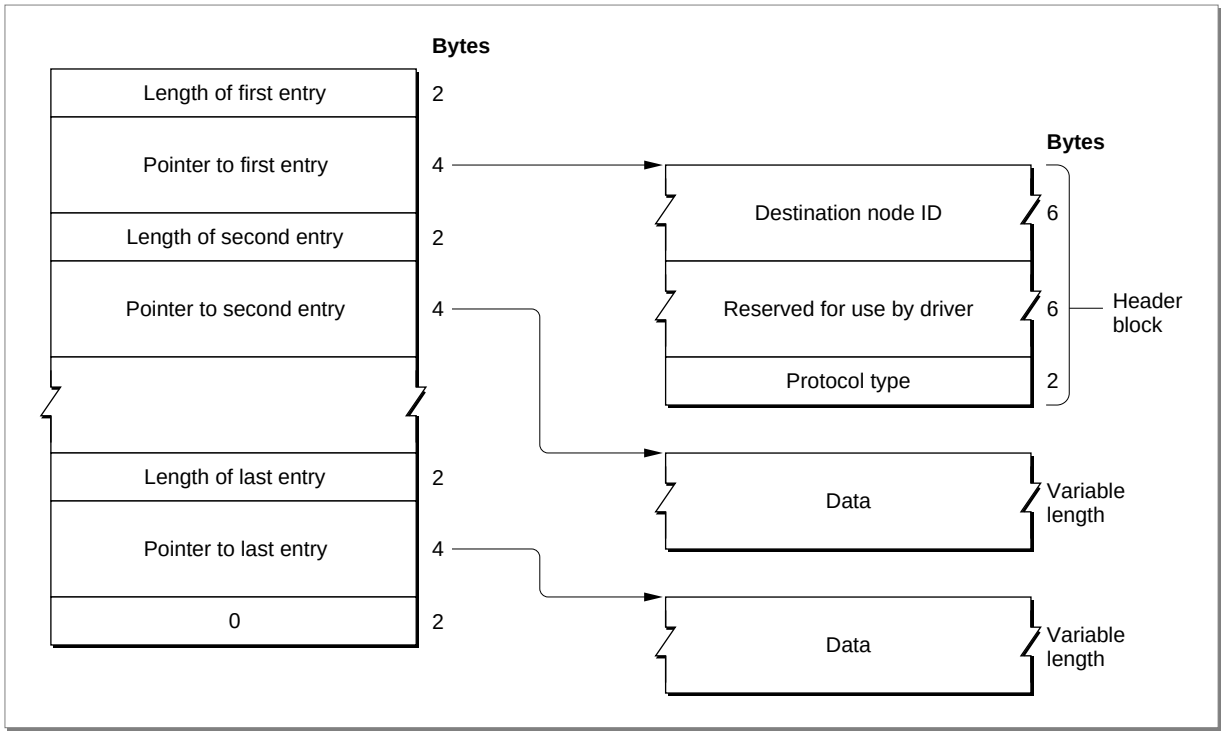
You use the `EWrite` function to send data to the .ENET driver for transmission over the Ethernet network. When you do this, you provide a pointer to a write-data structure containing one or more pairs of length words and pointers. (Figure 11-3 shows multiple pairs.) Each pair indicates the length and location of a portion of the data packet to be sent over the network. The first length-pointer pair points to a header block that is at least 14 bytes long and that starts with the destination node hardware address. Note that this is not the AppleTalk address, but is the *hardware* address of the destination node. (Note that this address can also be a multicast address or the broadcast address for the link type.)

The next 6 bytes of the header block are reserved for use by the .ENET driver. These bytes are followed by the 2-byte Ethernet protocol type field (Ethernet Phase 2 packets use this field to indicate the amount of data in the packet). Data may follow the header block; all other length-pointer pairs point to data. The write-data structure terminates with a 0 word.

Note
Instead of using multiple buffers and length-pointer pairs, you can create a write-data structure that consists of a single buffer that specifies the header block followed directly by the data. For more information about write-data structures, see the chapter "Datagram Delivery Protocol (DDP)" in this book. ◆

When you first open the .ENET driver, it allocates a 768-byte buffer that it uses for transmitting data packets. This buffer is large enough to hold the largest EtherTalk packet, which is 621 bytes in size. If you want to transmit data packets larger than 768 bytes, call the `ESetGeneral` function; the .ENET driver can then allocate a data buffer large enough to send packets up to 1514 bytes in size. Figure 11-3 shows the write-data structure that you use to send data to the .ENET driver.

Figure 11-3 An Ethernet write-data structure



The sample code in Listing 11-2 uses a multicast address instead of a local hardware address. The multicast address is a packet array that is defined as follows:

```
VAR
   gMultiCastAddr: PACKED ARRAY[0..5] OF Byte;
```

The following procedure initializes the `gMultiCastAddr` global variable:

```
PROCEDURE Init_Multicast_Address;
BEGIN
   gMultiCastAddr[1] := $09;
   gMultiCastAddr[2] := $00;
   gMultiCastAddr[3] := $2B;
   gMultiCastAddr[4] := $00;
   gMultiCastAddr[5] := $00;
   gMultiCastAddr[6] := $04;
END;
```

The code in Listing 11-2 defines an Ethernet write-data structure, and then it calls the `EWrite` function to send a data packet over Ethernet.

Listing 11-2 Sending a data packet over Ethernet

```
FUNCTION Send_Sample_ENET_Packet (ENETRefNum: Integer): OSErr;
CONST
   kSIZE1 = 100;
   kSIZE2 = 333;
TYPE
   WDS = RECORD               {write-data structure}
   length:  Integer;          {length of nth entry}
   aptr:    Ptr;              {pointer to nth entry}
END;

VAR
   myWDS:   ARRAY[1..4] OF WDS;
   myPB:    EParamBlock;      {.ENET parameter block}
   wheader: PACKED ARRAY[0..13] OF Byte;
   stuff1:  ARRAY[1..kSIZE1] OF Byte;
   stuff2:  ARRAY[1..kSIZE2] OF Byte;
   myErr:   OSErr;

BEGIN
   BlockMove(@gMultiCastAddr, @wheader, 6);  {multicast address}
   wheader[12] := $90;                       {protocol type}
   wheader[13] := $90;                       {must match kProtocol value}
   myWDS[1].length := 14;
   myWDS[1].aptr := @wheader;
   myWDS[2].length := kSIZE1;
   myWDS[2].aptr := @stuff1;
   myWDS[3].length := kSIZE2;
```

```
   myWDS[3].aptr := @stuff2;
   myWDS[4].length := 0;
   myPB.ePointer := @myWDS;
   myPB.ioRefNum := ENETRefNum;

{Send something.}
   myErr := EWrite(@myPB, FALSE);
   IF myErr <> noErr THEN
      DoError(myErr);
   Send_Sample_ENET_Packet := myErr;
END;
```

Using the Default Ethernet Protocol Handler to Read Data

This section describes how to write an application that uses the Apple default protocol handler for Ethernet Phase 1 packets. For Ethernet Phase 2 packets, the process is largely the same, except that you must code and provide your own protocol handler and use the LAP Manager to attach it.

When the Ethernet NuBus card or other Ethernet hardware receives a data packet, it generates an interrupt to the CPU. The interrupt handler in ROM determines the source of the interrupt and calls the .ENET driver. The .ENET driver reads the packet header to determine the protocol type of the data packet and checks to see if any client has specified that protocol type in a call to the `EAttachPH` function. If so, the client either specified a `NIL` pointer to a protocol handler or provided its own protocol handler. If the client specified a `NIL` pointer, the .ENET driver uses its default protocol handler to read the data. If no one has specified the protocol type that the packet header contains in a call to the `EAttachPH` function, the .ENET driver discards the data. (For more information about the `EAttachPH` function, see "EAttachPH" on page 11-28.)

The Ethernet driver looks for a pending `ERead` function with a protocol type that matches the packet protocol type. (When you call the `ERead` function, you pass it a protocol type.) The Ethernet driver places the entire packet—including the packet header—into the buffer specified by that function. The function returns the number of bytes actually read. If the packet is larger than the data buffer, the `ERead` function places as much of the packet as will fit into the buffer and returns the `buf2SmallErr` result code.

You must call the `ERead` function asynchronously to await the next data packet. When the .ENET driver receives the data packet, it completes execution of the `ERead` function and calls your completion routine. Your completion routine should call the `ERead` function again so that an `ERead` function is always pending execution. If the .ENET driver receives a data packet with a protocol type for which you specified the default protocol handler while no `ERead` function is pending, the .ENET driver discards the packet.

You can have several asynchronous calls to the `ERead` function pending execution simultaneously as long as you use different buffers and a different parameter block for each call.

Alternatively, after the `ERead` function completes execution, you can call the function again from your completion routine, and reuse the same parameter block. This is the approach the code in Listing 11-3 takes.

The code in Listing 11-3 calls the `EAttachPH` function to specify that the .ENET driver should use the default protocol handler to process packets for the protocol type defined by the following constant:

```
CONST
   kMyProtocol = $9090;             {must be > $5DC}
```

In practice, you should call the `EAttachPH` function very early, during your program initialization sequence, if possible. As soon as the connection is established and you are expecting data, you should call the `ERead` function asynchronously. The code in Listing 11-3 shows how to attach a protocol handler and read a packet for an Ethernet Phase 1 packet.

Listing 11-3 Attaching a protocol handler and reading a packet

```
FUNCTION Sample_AttachPH_And_Read_Packet (ENETRefNum: Integer): OSErr;
CONST
   kBigBytes = 8888;

VAR
   myPB:       MyEParamBlock;
   myEPBPtr:   MyEParamBlkPtr;
   aptr:       Ptr;
   myErr:      OSErr;

BEGIN
   myEPBPtr := @myPB;                          {set up EAttachPH parameters}
   WITH myPB.pb DO
      BEGIN
         eProtType := kMyProtocol;             {protocol type}
         ePointer := NIL;                      {use default protocol handler}
         ioRefNum := ENETRefNum;               {.ENET driver reference number}
      END;
   myErr := EAttachPH(EParamBlkPtr(myEPBPtr), FALSE);

   IF myErr <> noErr THEN                      {check if error occurred while }
      DoError(myErr)                           { attaching protocol handler}
   ELSE
      BEGIN
         aptr := NewPtr(kBigBytes);
         myPB.myA5 := SetCurrentA5;            {store the current A5 world}
```

```
    WITH myPB.pb DO
       BEGIN
          ioCompletion := @MyCompRoutine;
                                        {ptr to completion routine}
          eProtType := kMyProtocol;     {protocol type to respond to}
          ePointer := aptr;             {pointer to read-data buffer}
          eBuffSize := kBigBytes;       {size of read-data buffer}
          ioRefNum := ENETRefNum;       {.ENET driver refNum}
       END;
    myErr := ERead(EParamBlkPtr(myEPBPtr), TRUE);

    IF myErr <> noErr THEN
          {check if error occurred queueing read request}
       BEGIN
          DoError(myErr);               {process error result}
          Detach_SamplePH(ENETRefNum);  {detach protocol handler)
       END;
  END;
  Sample_AttachPH_And_Read_Packet := myErr;
END;
```

When the .ENET driver receives a packet, it then calls your completion routine if you called the `ERead` function asynchronously and the `ioCompletion` routine field is not `NIL`. Your completion routine should process the packet, after which it can then queue another asynchronous call to the `ERead` function to await the next packet.

The sample completion routine that Listing 11-4 shows uses the following inline function that gets the pointer to the parameter block from register A0.

```
FUNCTION GetParamBlockPtr: Ptr;
INLINE
   $2E88;         {MOVE.L A0,(SP)}
```

Because register A0 is a utility register that compilers often use for their own purposes, the sample code uses the following stub completion routine technique to minimize the possibility that a compiler will overwrite the value in register A0. The stub completion routine calls `GetParamBlockPtr` and then calls the actual completion routine.

```
PROCEDURE MyStubCompRoutine;

VAR
   myEPBPtr: MyEParamBlkPtr;

BEGIN
   myEPBPtr := MyEParamBlkPtr(GetParamBlockPtr);
               {get parameter block pointer from register A0}
   myCompRoutine(myEPBPtr);
               {now call the actual completion routine}
END;
```

Listing 11-4 shows the actual completion routine that the stub completion routine calls. This completion routine reuses the original parameter block when it calls the `ERead` function again. The code also shows how to access global variables from within the completion routine. Note that if you call the `ERead` function from within the completion routine, you must call the function asynchronously. You must not call the `ERead` function synchronously at interrupt time.

Listing 11-4 Completion routine to process received packet and await the next packet

```
PROCEDURE MyCompRoutine (myEPBPtr: MyEParamBlkPtr);
VAR
   myErr:   OSErr;
   saveA5:  LongInt;
   aptr: Ptr;

BEGIN
   saveA5 := SetA5(myEPBPtr^.myA5);    {set A5 to our world}
   IF (myEPBPtr^.pb.ioResult < noErr) THEN
                                       {was ERead successful?}
   BEGIN
      IF (myEPBPtr^.pb.ioResult <> reqAborted) THEN
                                       {was request aborted?}
         DoError(myEPBPtr^.pb.ioResult)
      END
   ELSE
   BEGIN                               {process the packet}
      aptr := myEPBPtr^.pb.EPointer;
      ProcessData(aptr);               {use the data}
   END;

  IF NOT gDone THEN                    {check if we have been called}
   BEGIN                               {if not, call ERead again}
      myErr := ERead(EParamBlkPtr(myEPBPtr), TRUE);
      IF myErr <> noErr THEN
         DoError(myErr);               {check if error occurred while }
                                       { queueing call to ERead}
   END;
   saveA5 := SetA5(saveA5);            {restore the A5 world}
END; {of MyCompletion routine}
```

Using Your Own Ethernet Protocol Handler to Read Data

If a client of the .ENET driver has used the `EAttachPH` function to provide a pointer to its own protocol handler, the .ENET driver calls that protocol handler, which must in turn call the .ENET driver's `ReadPacket` and `ReadRest` routines to read the data. Your protocol handler calls these routines in essentially the same way as you called these routines to implement a DDP socket listener. (The chapter "Datagram Delivery Protocol [DDP]" describes how you use these routines to implement a DDP socket listener.)

The following sections describe how the .ENET driver calls a custom protocol handler and the `ReadPacket` and `ReadRest` routines.

Note
Because an Ethernet protocol handler must read from and write to the CPU's registers, you must write the protocol handler in assembly language; you cannot write a protocol handler in Pascal. ◆

How the .ENET Driver Calls Your Protocol Handler

You can provide an Ethernet protocol handler for a particular protocol type and use the `EAttachPH` function to attach it to the .ENET driver. When the driver receives an Ethernet packet, it reads the packet header into an internal buffer, reads the protocol type, and calls the protocol handler for that protocol type. The CPU is in interrupt mode, and the registers are used as follows:

Registers on call to Ethernet protocol handler

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

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

A2 Free for your use

A3 Pointer to first byte past data-link header bytes (the first byte after the 2-byte 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 until after the `ReadRest` routine has completed execution

D0 Free for your use

D1 Number of bytes in the Ethernet packet left to be read (that is, the number of bytes following the Ethernet header)

D2 Free for your use

D3 Free for your use

If your protocol handler processes more than one protocol type, you can read the protocol type field in the frame header to determine the protocol type of the packet. The protocol-type field starts 2 bytes before the address pointed to by the A3 register.

Note
The source address starts 8 bytes before the address pointed to by the A3 register, and the destination address starts 14 bytes before the address pointed to by the A3 register. ◆

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 global variables.

How Your Protocol Handler Calls the .ENET Driver Routines

Your protocol handler must call the .ENET driver routines `ReadPacket` and `ReadRest` to read the incoming data packet.

Note
Before the Ethernet driver calls your protocol handler at interrupt time, you must have already allocated memory for one or more data buffers to hold the incoming data. ◆

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

Before you call the `ReadPacket` routine, you must place a pointer to the data buffer in the A3 register. You 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.

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 First byte after the last byte read into buffer

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

The `ReadPacket` routine indicates an error by clearing to 0 the zero (`z`) flag in the status register. If the `ReadPacket` routine returns an error, you must terminate execution of your protocol handler 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. 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, you can specify a buffer of size 0.

▲ **WARNING**
If you do not call the `ReadRest` routine after your last call to the `ReadPacket` routine, the system will crash. ▲

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 were read; 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)

The `ReadRest` routine indicates an error by clearing to 0 the zero (`z`) flag in the status register. You must terminate execution of your protocol handler with an RTS instruction whether or not the `ReadRest` routine returns an error.

Changing the Ethernet Hardware Address

Each Ethernet NuBus card or other Ethernet hardware interface device contains a unique 6-byte hardware address assigned by the manufacturer of the device. The .ENET driver normally uses this address to determine whether to receive a packet. To change the hardware address for your node, place in the System file a resource of type `'eadr'` with a resource ID equal to the slot number of the Ethernet NuBus card.

The `'eadr'` resource consists only of a 6-byte number. Do not use the broadcast address or a multicast address for this number. (Refer to *Inside AppleTalk,* second edition, for the broadcast and multicast address formats.)

When you open the .ENET driver, it looks for an `'eadr'` resource with the resource ID that matches the slot number of the card. If it finds one, the driver substitutes the number in this resource for the Ethernet hardware address and uses it until the driver is closed or reset.

Note
To avoid address collisions, you should never arbitrarily change the Ethernet hardware address. This feature should be used only by a system administrator who can keep track of all the Ethernet addresses in the system. ◆

Using the Token Ring Driver

You can write an application implementing a protocol other than AppleTalk that reads data from and writes it to the token ring driver defined by Apple.

To write data to the token ring driver and to perform other functions such as adding a functional address for the token ring hardware, you use the Ethernet functions described earlier, with the modifications noted later in this section. To read 802.2 packets from the token ring driver, you need to attach your protocol handler to the LAP Manager.

The Apple token ring driver implementation supports only the IEEE 802.2 Type 1 protocol and allows for the attachment of only one protocol handler that reads 802.2 packets that contain an SAP value of $AA.

Although it is possible to attach your own protocol handler at the hardware device driver level, Apple recommends that you not do this because it excludes AppleTalk from using the token ring driver. So that more than one protocol can receive packets from the token ring driver concurrently, Apple recommends that you attach your protocol handler to the LAP Manager. The LAP Manager attaches its own protocol handler to the token ring driver, and when it receives a packet for your protocol, the LAP Manager calls your protocol handler. When it receives a packet for another protocol, such as AppleTalk, the LAP Manager calls that application's protocol handler.

For a description of how to attach and detach your protocol handler for token ring, see the chapter "Link-Access Protocol (LAP) Manager" included in this book and the discussion of token ring and FDDI in "About Ethernet, Token Ring, and FDDI Support" beginning on page 11-3 in this chapter. The chapter "Link-Access Protocol (LAP) Manager" also gives more information on the SAP field value for 802.2 Type 1 packets.

Applying Ethernet Functions

The Apple token ring driver implements many but not all of the functions that the Apple Ethernet driver implements.

For those Ethernet functions that do apply to token ring, you use the function for token ring in the same way that you do for Ethernet: you pass parameters in a parameter block

and you use the Ethernet control code in the `csCode` field to call the function. The only difference is that instead of specifying the Ethernet driver reference number in the parameter block's `ioRefNum` field, you specify the token ring driver reference number. Here are the Ethernet functions that apply to token ring:

- You use the `EAddMulti` function to add a functional address for token ring and the `EDelMulti` function to remove one. Be careful not to specify the broadcast address as a functional address. See *Inside AppleTalk*, second edition, for a description and the format of functional and broadcast addresses for token ring.
- You use the `EWrite` function to send data to the token ring driver for transmission over the network.

Here are the Ethernet functions that do not apply to token ring:

- The `ERead` and `ERdCancel` functions are not valid for token ring because Apple does not specify a default protocol handler for the token ring driver. These two functions are used exclusively by applications that use the default Ethernet protocol handler. If an application calls these functions for token ring, the driver will return an error.
- The `ESetGeneral` function switches to a mode that allows the .ENET driver to transmit a larger Ethernet data packet than the standard size. Because token ring is not normally restricted to the limited packet size, this function does not apply. However, the token ring driver will return a result of `noErr` if you call this function.

There are some other differences between Ethernet and token ring:

- The token ring packet size is determined by the token ring hardware developer. However, for Logical Link Control (LLC) type packets, the packet length cannot exceed 1500 bytes.
- The token ring interface uses functional addresses instead of multicast addresses. Be careful not to use the broadcast address for a functional address. For information about both kinds of token ring addresses, see *Inside AppleTalk*, second edition.
- For token ring, the vendor who supplies the hardware device driver provides a control panel that allows you to specify an alternative hardware address. (For general information about alternative hardware addresses, see "Changing the Ethernet Hardware Address" on page 11-19.)

Note

Although you can use the `EAttachPH` function to attach a protocol handler to the token ring driver and the `EDetachPH` function to remove one, Apple recommends that you not use these functions. Instead, you should use the LAP Manager's `L802Attach` and `L802Detach` routines. ◆

Sending and Receiving Data

The tasks involved in sending data to and receiving it from a token ring driver are similar to those that you use for Ethernet. The primary difference is that you use the LAP Manager to attach your protocol handler. Any vendor implementing a token ring driver to run on a Macintosh computer must follow rules that direct them to return packet information in the same manner as does the Ethernet driver for 802.2 packets. From the

perspective of an application that uses the token ring driver, this means that when the LAP Manager calls your protocol handler, you can expect the token ring hardware addresses that you reference from register A3 to follow the same format that is used for Ethernet addresses, regardless of how the token ring address might appear at the hardware level.

Here are the steps that you follow to send data to and receive it from a token ring driver:

1. Locate the token ring cards that are installed in the system. Use the Slot Manager to identify installed token ring cards. Use the `SGetTypeSRsrc` function described in the Slot Manager chapter of *Inside Macintosh: Devices* to determine which NuBus slots contain token ring cards. To find token ring cards, use the value `catNetwork (0x4)` in the `spCategory` field and the value `typeTokenRing (0x2)` in the `spCType` field. You should provide a user interface that allows the user to select a specific token ring card in the case that more than one is present.
2. Use the `OpenSlot` function to open the token ring driver. Set the `ioNamePtr` field to .TOKN. If you did not locate any NuBus token ring cards in step 1, you should also attempt to open the .TOKN0 driver in case non-NuBus token ring hardware is attached to the system. Use the Device Manager's `OpenDriver` function to open the .TOKN0 driver. (For information on the `OpenSlot` and `OpenDriver` functions, see the chapter "Device Manager" in *Inside Macintosh: Devices*.)

   Note that this section refers to the .TOKN driver shell, which facilitates multi-vendor support, as the .TOKN driver. Opening the .TOKN driver shell, which loads and opens the card's driver, is effectively the same as directly opening the token ring driver.
3. If your application requires a functional address, use the `EAddMulti` function to register one. Functional addresses are the token ring equivalent of Ethernet and FDDI multicast addresses. (For information on functional addresses, see *Inside AppleTalk,* second edition. For a description of multicast addresses, see "About Multicast Addressing" on page 11-7.)
4. Use the LAP Manager's `L802Attach` routine to install your protocol handler. (See the chapter "Link-Access Protocol [LAP] Manager" in this book for more information.)
5. Use the `EWrite` function to send packets to the token ring driver for transmission across the network. To use the `EWrite` function, you provide a pointer to a write-data structure. The first buffer in the write-data structure must be at least 14 bytes long: the first 6 bytes of that buffer must contain the destination address. Bytes 13 and 14 must contain the packet length, which must not exceed 1500 bytes. The token ring driver fills in bytes 7–12 with the source address. (For more information on the write-data structure, see "Using a Write-Data Structure to Transmit Ethernet Data" on page 11-10.)
6. When you are finished using the token ring driver, use the LAP Manager's `L802Detach` routine to remove your protocol handler.
7. When you are finished using a functional address, use the `EDelMulti` function to remove it.

Using the FDDI Driver

You can write an application implementing a protocol other than AppleTalk that processes 802.2 packets and that sends and receives data over a Fiber Distributed Data Interface (FDDI) network. To do this, you read data from and write it to the FDDI driver defined by Apple. Your application can run on a node that is also running AppleTalk.

To write data to the FDDI driver and to perform other functions such as adding a multicast address for the FDDI hardware, you use the Ethernet functions described earlier in this chapter. To receive 802.2 packets from the FDDI driver, you attach your protocol handler to the LAP Manager using the interface to the LAP Manager.

The Apple FDDI driver implementation support allows for the attachment of only one protocol handler. The Apple FDDI driver specification requires that an FDDI driver handle 802.2 packets to service access points (SAP) other than SAP $AA.

Although it is possible to attach your own protocol handler at the hardware device driver level, Apple Computer, Inc. recommends that you not do this because it excludes AppleTalk from using the FDDI driver. So that more than one protocol can receive packets from the FDDI driver concurrently, Apple recommends that you attach your protocol handler to the LAP Manager. The LAP Manager attaches its own protocol handler to the FDDI driver, and when it receives a packet for your protocol, the LAP Manager calls your protocol handler. When it receives a packet for another protocol, such as AppleTalk, the LAP Manager calls that application's protocol handler.

For a description of how to attach and detach your protocol handler for FDDI, see the chapter "Link-Access Protocol (LAP) Manager" included in this book and the discussion of token ring and FDDI in "About Ethernet, Token Ring, and FDDI Support" beginning on page 11-3 in this chapter. The chapter "Link-Access Protocol (LAP) Manager" also explains the concept and use of the SAP field value for 802.2 Type 1 packets.

Applying Ethernet Functions

The Apple FDDI driver implements many but not all of the functions that the Apple Ethernet driver implements.

For those Ethernet functions that do apply to FDDI, you use the function for FDDI in the same way that you do for Ethernet: you pass parameters in a parameter block and you use the Ethernet control code in the `csCode` field to call the function. The only difference is that instead of specifying the Ethernet driver reference number in the parameter block's `ioRefNum` field, you specify the FDDI driver reference number. Here are the Ethernet functions that apply to FDDI:

- You use the `EAddMulti` function to add a multicast address for FDDI and the `EDelMulti` function to remove one. Be careful not to use the broadcast address as a multicast address. The broadcast and multicast addresses are the same for FDDI and Ethernet. For information about these addresses and their formats, see the discussion of them for Ethernet in *Inside AppleTalk*, second edition.
- You use the `EWrite` function to send data to the FDDI driver for transmission over the network.

Here are the Ethernet functions that do not apply to FDDI:

- The `ERead` and `ERdCancel` functions are not valid for FDDI because Apple does not specify a default protocol handler for the FDDI driver. These two functions are used exclusively by applications that use the default Ethernet protocol handler. If an application calls these functions for FDDI, the driver will return an error.
- The `ESetGeneral` function switches to a mode that allows the .ENET driver to transmit a larger Ethernet data packet than the standard size. Because FDDI is not normally restricted to the limited packet size, this function does not apply. However, the FDDI driver will return a result of `noErr` if you call this function.

There are some other differences between Ethernet and FDDI:

- The FDDI packet size is determined by the FDDI hardware developer. However, for Logical Link Control (LLC) type packets, the packet length cannot exceed 1500 bytes.
- The FDDI driver searches for a resource of type `'fadr'` instead of `'eadr'` in the System file for an alternative hardware address. (For general information about alternative hardware addresses, see "Changing the Ethernet Hardware Address" on page 11-19.)

Note

Although you can use the `EAttachPH` function to attach a protocol handler to the FDDI driver and the `EDetachPH` function to remove one, Apple recommends that you not use these functions. Instead, you should use the LAP Manager's `L802Attach` and `L802Detach` routines. ◆

Sending and Receiving Data

The tasks involved in sending data to and receiving it from an FDDI driver are similar to those that you use for Ethernet. The primary difference is that you use the LAP Manager to attach your protocol handler. Any vendor implementing an FDDI driver to run on a Macintosh computer must follow rules that direct them to return packet information in the same manner as does the Ethernet driver for 802.2 packets. From the perspective of an application that uses the FDDI driver, this means that when the LAP Manager calls your protocol handler, you can expect the FDDI hardware addresses that you reference from register A3 to follow the same format that is used for Ethernet addresses, regardless of how the FDDI address might appear in the packet. The chapter "Link-Access Protocol (LAP) Manager" in this book explains this in detail.

Here are the steps that you follow to send data to and receive it from an FDDI driver:

1. Locate the FDDI cards that are installed in the system. Use the Slot Manager to identify installed FDDI cards. Use the `SGetTypeSRsrc` function described in the Slot Manager chapter of *Inside Macintosh: Devices* to determine which NuBus slots contain FDDI cards. To find FDDI cards, use the value `catNetwork (0x4)` in the `spCategory` field and the value `typeFDDI (0x11)` in the `spCType` field. You should provide a user interface that allows the user to select a specific FDDI card in the case that more than one is present.
2. Use the `OpenSlot` function to open the FDDI driver. Set the `ioNamePtr` field to .FDDI. If you did not locate any NuBus FDDI cards in step 1, you should also attempt to open the .FDDI0 driver in case non-NuBus FDDI hardware is attached to the

system. Use the Device Manager's `OpenDriver` function to open the .FDDI0 driver. (For information on the `OpenSlot` and `OpenDriver` functions, see the chapter "Device Manager" in *Inside Macintosh: Devices*.)

Note that this section refers to the .FDDI driver shell, which facilitates multivendor support, as the .FDDI driver. Opening the .FDDI driver shell, which loads and opens the card's driver, is effectively the same as directly opening the FDDI driver.

3. If your application requires a multicast address, use the `EAddMulti` function to register a multicast address. (For information on multicast addresses, see *Inside AppleTalk*, second edition. For a description of multicast addresses, see "About Multicast Addressing" on page 11-7.)
4. Use the LAP Manager's `L802Attach` routine to install your protocol handler. (See the chapter "Link-Access Protocol [LAP] Manager" in this book for more information.)
5. Use the `EWrite` function to send packets to the FDDI driver for transmission across the network. To use the `EWrite` function, you provide a pointer to a write-data structure. The first buffer in the write-data structure must be at least 14 bytes long: the first 6 bytes of that buffer must contain the destination address. Bytes 13 and 14 must contain the packet length, which must not exceed 1500 bytes. The FDDI driver fills in bytes 7–12 with the source address. (For more information on the write-data structure, see "Using a Write-Data Structure to Transmit Ethernet Data" on page 11-10.)
6. When you are finished using the FDDI driver, use the LAP Manager's `L802Detach` routine to remove your protocol handler.
7. When you are finished using a multicast address, use the `EDelMulti` function to remove it.

Ethernet, Token Ring, and FDDI Reference

This section describes the Ethernet data structures and functions. You use these data structures and functions to communicate directly with the Ethernet, token ring, and FDDI drivers. The functions were originally designed to read data from and write it to the Ethernet driver. However, by specifying the appropriate driver reference number, you can also use many of these functions for the token ring and FDDI drivers.

Some of the Ethernet functions do not apply to token ring and FDDI. Each of the functions includes a section called *Token Ring and FDDI Considerations* that identifies whether the function is valid for these drivers.

The "Data Structures" section shows the Pascal data structures for the write-data structure and the ENET parameter block of type `EParamBlock`.

The "Routines" section describes how to

- attach and detach a protocol handler to receive data from an Ethernet driver
- write data to the Ethernet, token ring, or FDDI driver
- read data from the Ethernet driver and cancel a function request to read data from the driver when you use the default Ethernet protocol handler

- obtain information about the Ethernet driver and switch its mode to handle larger packets
- add and remove a multicast address for an application that uses the Ethernet or FDDI driver and a functional address for an application that uses the token ring driver

Data Structures

This section describes the data structures that you use to provide information to the Ethernet, token ring, and FDDI drivers. You use the write-data structure to provide the addressing information and data to send to another node over the network. You use the ENET parameter block of type `EParamBlock` to pass information to and receive it from the functions for Ethernet, token ring, and FDDI drivers.

The Write-Data Structure

To send data directly from the Ethernet, token ring, or FDDI driver, you must provide a write-data structure and pass the `EWrite` function a pointer to it. A write-data structure contains a series of pairs of length words and pointers. Each pair indicates the length and location of a portion of the data that constitutes the packet to be sent over the network. The interface files for the driver do not include a type declaration for the write-data structure. Here is an example type declaration that you can include in your application.

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

Field descriptions

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

`entryPtr` A pointer to the data that is part of the packet to be sent using the `EWrite` function.

For more information about the write-data structure, see "Using a Write-Data Structure to Transmit Ethernet Data" beginning on page 11-10.

The Parameter Block for Ethernet, Token Ring, and FDDI Driver Functions

All of the driver functions—`EAttachPH`, `EDetachPH`, `EWrite`, `ERead`, `ERdCancel`, `EGetInfo`, `ESetGeneral`, `EAddMulti`, `EDelMulti`—require a pointer to an ENET parameter block of type `EParamBlock`.

This section defines the fields that are common to all of the driver functions that use the ENET parameter block. The ENET parameter block contains reserved fields that are used internally by the .ENET driver; these fields are not described. The fields that are used for specific functions only are defined in the descriptions of the functions to which they apply.

```
TYPE EParamBlock =
   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 Integer OF
         ENetWrite, ENetAttachPH, ENetDetachPH, ENetRead,
         ENetRdCancel,ENetGetInfo,ENetSetGeneral:
            (eProtType:    Integer;       {Ethernet protocol type}
             ePointer:     Ptr;           {pointer; use depends on function}
             eBuffSize:    Integer;       {buffer size}
            eDataSize:     Integer);      {number of bytes read}
         ENetAddMulti,ENetDelMulti:
            (eMultiAddr:   ARRAY[0..5] OF Char;)   {multicast address}
      END;
```

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the system 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.

`ioResult` The result of the function. If you call the function asynchronously, the function sets this field to 1 as soon as it begins execution, and it changes the field to the actual result code when it completes execution.

`ioRefNum` The driver reference number that the `OpenDriver` function or the `OpenSlot` function returns.

`csCode` A routine selector for the function to be executed. Each function has a unique routine selector. The MPW interface automatically sets this value for you.

Routines

An application that uses AppleTalk Manager routines for network communication can communicate with whatever AppleTalk network the user has selected through the Network control panel. However, you can choose to write an application that talks only to the hardware device driver for a particular type of network, such as Ethernet; in this case, your application has to address the hardware driver directly. This section describes the functions that you use to

- attach a protocol handler to the .ENET driver
- detach a protocol handler that you previously attached
- send data directly to a hardware device driver
- read data from the .ENET driver
- cancel a pending call to read data from the .ENET driver
- obtain information about the .ENET driver
- switch the .ENET driver mode
- add a multicast or functional address
- remove a multicast or functional address

Attaching and Detaching an Ethernet Protocol Handler

You can use the functions that this section describes to attach a protocol handler to the .ENET driver, to specify which protocol handler the .ENET driver is to use for each protocol type, and to detach a protocol handler that you previously attached.

Note
Apple Computer, Inc. recommends that you attach a protocol handler for a token ring or an FDDI driver using the interface to the LAP Manager. ◆

EAttachPH

The `EAttachPH` function attaches a protocol handler to the .ENET driver to receive packets of a particular protocol type. You can provide and attach your own protocol handler or use the default protocol handler provided by Apple.

```
FUNCTION EAttachPH (thePBptr: EParamBlkPtr;
                    async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

`async` A Boolean value that specifies whether the function is to be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to completion routine.
←	ioResult	OSErr	The result code.
→	ioRefNum	Integer	The driver reference number.
→	csCode	Integer	Always ENetAttachPH for this function.
→	eProtType	Integer	The Ethernet protocol type.
→	ePointer	Ptr	A pointer to protocol handler.

Field descriptions

`eProtType` The protocol type for which you are attaching a protocol handler. To attach a protocol handler for Ethernet Phase 1 packets, specify 0 as the value of this field. (Ethernet Phase 1 packets are IEEE 802.3 protocol packets.)

`ePointer` A pointer to your protocol handler application. To use the default protocol handler that Apple provides, set this field value to `NIL`.

DESCRIPTION

The `EAttachPH` function serves two purposes: you can use it to attach to the .ENET driver your own protocol handler for a specific protocol type, or you can use it to specify that the .ENET driver should call the default protocol handler for your protocol type. If you attach your own protocol handler, the .ENET driver calls that protocol handler each time it receives a packet with the protocol type you specified. If you specify that the .ENET driver should use the default protocol handler, then you use the `ERead` command to read packets with that protocol type. In practice, you should call the `EAttachPH` function very early, during your program initialization sequence, if possible.

You specify the protocol type in the `eProtType` parameter and provide a pointer to the protocol handler in the `ePointer` parameter. If you specify `NIL` for the `ePointer` parameter, then the .ENET driver uses the default protocol handler for that protocol type.

SPECIAL CONSIDERATIONS

Instead of using the `EAttachPH` function to install a protocol handler for an Ethernet Phase 2 packet, you should use the LAP Manager's `L802Attach` routine. In the case of an 802.3 protocol packet, the .ENET driver passes the packet to the LAP Manager 802.2 protocol handler. If the packet has the protocol type you specified with the `L802Attach` routine, the LAP Manager passes the packet on to your protocol handler. For information about the `L802Attach` routine, see the chapter "Link-Access Protocol (LAP) Manager" in this book.

TOKEN RING AND FDDI CONSIDERATIONS

This function is available for token ring and FDDI also. However, Apple Computer, Inc. recommends that you use the LAP Manager's `L802Attach` routine instead to attach your protocol handlers for token ring and FDDI. For information about the `L802Attach` routine, see the chapter "Link-Access Protocol (LAP) Manager" in this book.

Note that if you use this function for token ring or FDDI, you exclude other processes, such as AppleTalk, from attaching their protocol handlers to the driver at the same time. If you use the LAP Manager interface, other applications can also attach their protocol handlers and use the driver concurrently.

If you use this function for token ring, you can only install a protocol handler for protocol type 0. To use this function for either token ring or FDDI, you must set the `ioRefNum` field to the driver reference number that the `OpenSlot` or the `OpenDriver` function returns.

Apple does not provide a default protocol handler for token ring or FDDI.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EAttachPH` function from assembly language, call the `_Control` trap macro with a value of `ENetAttachPH` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`LAPProtErr`	–94	Protocol handler is already attached or node's protocol table is full

SEE ALSO

For more information on how to use the `EAttachPH` function, see "Using the Default Ethernet Protocol Handler to Read Data" beginning on page 11-13.

For information on the IEEE 802.2 and 802.3 protocols, see the chapter "Link-Access Protocol (LAP) Manager" in this book.

EDetachPH

The `EDetachPH` function detaches a protocol handler from the .ENET driver.

```
FUNCTION EDetachPH (thePBptr: EParamBlkPtr;
                    async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	ioCompletion	ProcPtr	A pointer to completion routine.
←	ioResult	OSErr	The result code.
→	ioRefNum	Integer	The driver reference number.
→	csCode	Integer	Always ENetDetachPH for this function.
→	eProtType	Integer	The Ethernet protocol typ.

Field descriptions

`eProtType` The protocol type whose protocol handler you want to remove.

DESCRIPTION

You use the `EDetachPH` function to remove from the .ENET driver a protocol handler that you attached using the `EAttachPH` function. When you call the `EDetachPH` function to remove the protocol handler, `EDetachPH` removes the protocol type from the node's protocol table. Once the protocol type is removed from the node's table, the .ENET driver no longer delivers packets with that protocol type. You specify the protocol type in the `eProtType` parameter.

If you specified your protocol type and attached the default protocol handler, `EDetachPH` removes the entry from the node's protocol table. When you call the `EDetachPH` function, any pending calls to the `ERead` function terminate with the `reqAborted` result code.

TOKEN RING AND FDDI CONSIDERATIONS

This function is available for token ring and FDDI also. However, Apple Computer, Inc. recommends that you use the LAP Manager interface to attach and detach a protocol handler for token ring and FDDI. To detach a protocol handler, you use the LAP Manager's `L802Detach` routine. For information about the `L802Detach` routine, see the chapter "Link-Access Protocol (LAP) Manager" in this book.

Note that if you use this function for token ring or FDDI, you must set the `ioRefNum` field to the driver reference number that the `OpenSlot` or `OpenDriver` function returns. For token ring, you can only detach a protocol handler for protocol type 0.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EDetachPH` function from assembly language, call the `_Control` trap macro with a value of `ENetDetachPH` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`LAPProtErr`	–94	No protocol handler is attached

Writing and Reading Ethernet Packets

You can use the functions in this section to send data to an Ethernet, token ring, or FDDI driver to be transmitted over the network. When you use the default Ethernet protocol handler, you can use the `ERead` and `ERdCancel` functions to read Ethernet packets and cancel execution of a read operation.

EWrite

The `EWrite` function allows you to send data directly to a hardware device driver for a particular network type for transmission across the network.

```
FUNCTION EWrite (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `ENetWrite` for this function.
→	`ePointer`	`Ptr`	A pointer to write-data structure.

Field descriptions

`ePointer` A pointer to the write-data structure that contains the data that you want to send.

DESCRIPTION

You use the `EWrite` function to send a data packet over an Ethernet, a token ring, or an FDDI network by communicating directly with the hardware device driver for that network type. You must first prepare a write-data structure that specifies the destination address and the protocol type and contains the data that you want to send. You place a pointer to the write-data structure in the `ePointer` parameter.

For Ethernet, if you want to send a packet larger than 768 bytes, you must first call the `ESetGeneral` function to put the .ENET driver in general-transmission mode. If the size of the packet you provide is less than 60 bytes, the driver adds pad bytes to the packet.

TOKEN RING AND FDDI CONSIDERATIONS

You can use this function to send data to a token ring or FDDI driver. Note that the packet size for token ring and FDDI is hardware dependent. However, for Logical Link Control (LLC) type packets, the packet length cannot exceed 1500 bytes.

To use this function for token ring or FDDI, you must set the `ioRefNum` field to the driver reference number that the `OpenSlot` or `OpenDriver` function returns.

You must also provide a pointer to a write-data structure. The first buffer in the write-data structure must be at least 14 bytes long: the first 6 bytes of that buffer must contain the destination address. Bytes 13 and 14 must contain the packet length, which must not exceed 1500 bytes. The token ring driver fills in bytes 7–12 with the source address.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EWrite` function from assembly language, call the `_Control` trap macro with a value of `ENetWrite` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`eLenErr`	–92	Packet too large or first entry of the write-data structure did not contain the full 14-byte header
`excessCollsns`	–95	Hardware error

SEE ALSO

For information on how to use the `EWrite` function and how to create a write-data structure, see "Using a Write-Data Structure to Transmit Ethernet Data" beginning on page 11-10.

ERead

When you use the default protocol handler for Ethernet that Apple provides, you must use the `ERead` function to read a data packet and place it in a data buffer.

```
FUNCTION ERead (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

`thePBptr`	A pointer to a parameter block of type `EParamBlock`.
`async`	A Boolean value that specifies whether the function should be executed asynchronously or synchronously. Specify `TRUE` for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to completion routine.
←	ioResult	OSErr	The result code.
→	ioRefNum	Integer	The driver reference number.
→	csCode	Integer	Always ENetRead for this function.
→	eProtType	Integer	The Ethernet protocol type.
→	ePointer	Ptr	A pointer to a data buffer.
→	eBuffSize	Integer	The size of the data buffer.
←	eDataSize	Integer	The number of bytes read.

Field descriptions

eProtType	The protocol type of the packet you want to read.
ePointer	A pointer to the data buffer into which you want to read data.
eBuffSize	The size of the data buffer. If you are expecting Ethernet data packets, the buffer should be at least 621 bytes in size; if you are expecting general Ethernet data packets, the buffer should be at least 1514 bytes in size.
eDataSize	The number of bytes of data actually read.

DESCRIPTION

You can use the ERead function to read packets of a particular protocol type only after you have used the EAttachPH function to specify a NIL pointer to the protocol handler to indicate that you want to use the default protocol handler. In practice, you should call the EAttachPH function very early, during your program initialization sequence, if possible. As soon as the connection is established and you are expecting data, you should call the ERead function asynchronously.

The ERead function places the entire packet, including the packet header, into your buffer. The function returns in the eDataSize parameter the number of bytes actually read. If the packet is larger than the data buffer, the ERead function places as much of the packet as will fit into the buffer and returns the buf2SmallErr result code.

Call the ERead function asynchronously to await the next data packet. When the .ENET driver receives the data packet, it completes execution of the ERead function and calls your completion routine. If the .ENET driver receives a data packet with a protocol type for which you specified the default protocol handler while no ERead command is pending, the driver discards the data packet.

You can have several asynchronous calls to the ERead function pending execution simultaneously as long as you use a different parameter block for each call.

SPECIAL CONSIDERATIONS

You must not use the ERead function to read packets if you supply and attach your own protocol handler. In this case, you use the driver's ReadPacket and ReadRest routines from within your protocol handler.

TOKEN RING AND FDDI CONSIDERATIONS

This function does not apply to token ring and FDDI.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `ERead` function from assembly language, call the `_Control` trap macro with a value of `ENetRead` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`LAPProtErr`	–94	No protocol is attached or protocol handler pointer was not 0
`reqAborted`	–1105	`ERdCancel` or `EDetachPH` function called
`buf2SmallErr`	–3101	Packet too large for buffer; partial data returned

SEE ALSO

See "Using the Default Ethernet Protocol Handler to Read Data" beginning on page 11-13 for more information on using the `ERead` function.

ERdCancel

The `ERdCancel` function cancels execution of a specific call to the `ERead` function.

```
FUNCTION ERdCancel (thePBptr: EParamBlkPtr;
                    async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `ENetRdCancel` for this function.
→	`ePointer`	`Ptr`	A pointer to `ERead` parameter block.

Field descriptions

`ePointer` A pointer to the .ENET parameter block that you specified when you called the `ERead` function that you want to cancel.

DESCRIPTION

To cancel an `ERead` function request using the `ERdCancel` function, you must have called the `ERead` function asynchronously. You specify in the `ePointer` parameter a pointer to the parameter block that you used when you called the `ERead` function.

When you call the `ERdCancel` function, the pending `ERead` function that you cancel receives the `reqAborted` result code.

TOKEN RING AND FDDI CONSIDERATIONS

This function is not valid for token ring and FDDI.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `ERdCancel` function from assembly language, call the `_Control` trap macro with a value of `ENetRdCancel` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`cbNotFound`	–1102	`ERead` not active

Obtaining Information About the Ethernet Driver and Switching Its Mode

The functions in this section return information about the .ENET driver and switch the .ENET driver from limited-transmission mode to general-transmission mode.

EGetInfo

The `EGetInfo` function returns information about the .ENET driver.

```
FUNCTION EGetInfo (thePBptr: EParamBlkPtr;
                   async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `ENetGetInfo` for this function.
→	`ePointer`	`Ptr`	A pointer to a buffer.
→	`eBuffSize`	`Integer`	The size of the buffer.
←	`eDataSize`	`Integer`	The number of bytes returned.

Field descriptions

`ePointer`	A pointer to a buffer that is at least 18 bytes in size. The `EGetInfo` function returns the information about the .ENET driver in this buffer.
`eBuffSize`	The size of the buffer pointed to by `ePointer`.
`eDataSize`	The number of bytes of information that `EGetInfo` returns in the buffer pointed to by `ePointer`.

DESCRIPTION

The `EGetInfo` function returns information about the .ENET driver. Beginning with version 58 of AppleTalk, the `EGetInfo` function returns additional information for SONIC-based network interface controllers (NICs). For these cards, `EGetInfo` can return up to 78 bytes of information. The `eDataSize` field returns the number of bytes of information that `EGetInfo` has placed in the data buffer that you provide. You can use the value returned in this field to determine whether or not the Ethernet card uses a SONIC chip. For all cards that are not SONIC based, this field will contain a value of 18.

If you do not know whether the Ethernet card that you are using has a SONIC chip, you should provide a data buffer that is at least 78 bytes in length. If you are certain that the Ethernet card that you are using is not SONIC based, you must provide a data buffer that is at least 18 bytes. Put a pointer to the buffer in the `ePointer` parameter and the size of the buffer in the `eBuffSize` parameter.

For Ethernet cards that are not SONIC based, the `EGetInfo` function places the following information in the data buffer:

Bytes	Information
1–6	Ethernet address of the node on which the driver is installed
7–10	Number of times the receive queue has overflowed
11–14	Number of data transmission operations that have timed out
15–18	Number of packets received that contain an incorrect address

An incorrect Ethernet address is one that is neither the broadcast address, a multicast address for which this node is registered, nor the node's data-link address. A node could receive an incorrect Ethernet address due to a hardware or software error.

For SONIC-based Ethernet cards, the last 60 bytes in the buffer return information from the SONIC chip network statistic counters. The `EGetInfo` function places the following information in the data buffer:

Bytes	Information
1–6	Ethernet address of the node on which the driver is installed
7–10	No information returned (zero-filled)
11–14	No information returned (zero-filled)
15–18	No information returned (zero-filled)
19–22	Frames transmitted without error
23–26	Single collision frames
27–30	Multiple collision frames
31–34	Collision frames
35–38	Frames with deferred transmission
39–42	Late collision
43–46	Excessive collisions
47–50	Excessive deferrals
51–54	Internal MAC transmit error
55–58	Frames received without error
59–62	Multicast frames received without error
63–66	Broadcast frames received without error
67–70	Frame check sequence errors
71–74	Alignment errors
75–78	Frames lost due to internal MAC receive errors

TOKEN RING AND FDDI CONSIDERATIONS

This function does not apply to token ring and FDDI.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EGetInfo` function from assembly language, call the `_Control` trap macro with a value of `ENetGetInfo` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error

ESetGeneral

The `ESetGeneral` function switches the .ENET driver from limited-transmission mode to general-transmission mode, allowing it to transmit a larger data packet.

```
FUNCTION ESetGeneral (thePBptr: EParamBlkPtr;
                      async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `ENetSetGeneral` for this function.

DESCRIPTION

The `ESetGeneral` function switches the .ENET driver from limited-transmission mode to general-transmission mode, which enables the .ENET driver to transmit an Ethernet data packet of up to 1514 bytes. In limited-transmission mode, the .ENET driver allocates a write-data buffer of 768 bytes. This buffer size is more than sufficient to hold an Ethernet data packet, which can be no larger than 621 bytes. However, if you want to send a packet that is larger than the Ethernet data packet, you must use the general-transmission mode.

SPECIAL CONSIDERATIONS

There is no command to switch the .ENET driver from general-transmission mode to limited-transmission mode. To switch back to limited-transmission mode, you have to reset the driver by restarting the computer.

TOKEN RING AND FDDI CONSIDERATIONS

This function does not apply to token ring and FDDI. However, if an application calls this function for token ring or FDDI, the driver will return a value of `noErr` in register D0.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `ESetGeneral` function from assembly language, call the `_Control` trap macro with a value of `ENetSetGeneral` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Insufficient memory in heap

Adding and Removing Ethernet Multicast Addresses

The functions in this section add or delete multicast addresses for Ethernet or FDDI for a particular node and functional addresses for token ring for a particular node.

EAddMulti

The `EAddMulti` function adds a multicast address or a functional address to the node that is running your application.

```
FUNCTION EAddMulti (thePBptr: EParamBlkPtr;
                    async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	Result code.
→	`ioRefNum`	`Integer`	Driver reference number.
→	`csCode`	`Integer`	Always `ENetAddMulti` for this function.
→	`eMultiAddr`	`6-byte array`	Multicast address.

Field descriptions

`eMultiAddr` The multicast address that you want to add and use.

DESCRIPTION

You use the `EAddMulti` function to add a multicast address for Ethernet or FDDI to the node that is running your application so that the hardware device driver for that network type will accept packets delivered to that address. You can also use this function to add a functional address that serves the same purpose for token ring.

Each time a client of a hardware device driver calls the `EAddMulti` function for a particular multicast address, the driver increments a counter for that multicast address. Each time a client of the hardware device driver calls the `EDelMulti` function, the driver decrements the counter for that address. As long as the count for a multicast address is equal to or greater than 1, the hardware device driver accepts packets directed to that multicast address. Therefore, if any client of the hardware device driver in the node has called the `EAddMulti` function for a particular multicast address, the driver receives packets delivered to that address. This process also applies to token ring for functional addresses. For information on how to specify multicast and functional addresses, see *Inside AppleTalk,* second edition. Be careful not to use the broadcast address, which is also described in *Inside AppleTalk,* as a functional address.

TOKEN RING AND FDDI CONSIDERATIONS

If your token ring application requires a functional address, use the `EAddMulti` function to register a functional address. Functional addresses are the token ring equivalent of Ethernet and FDDI multicast addresses. If your FDDI application requires a multicast address, use the `EAddMulti` function to register a multicast address.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EAddMulti` function from assembly language, call the `_Control` trap macro with a value of `ENetAddMulti` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`eMultiErr`	–91	Invalid address or table is full

EDelMulti

The `EDelMulti` function decrements the counter kept by the hardware device driver for a particular multicast address for Ethernet or FDDI or a particular functional address for token ring.

```
FUNCTION EDelMulti (thePBptr: EParamBlkPtr;
                    async: Boolean): OSErr;
```

`thePBptr` A pointer to a parameter block of type `EParamBlock`.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `ENetDelMulti` for this function.
→	`eMultiAddr`	`6-byte array`	A multicast address.

Field descriptions

`eMultiAddr` The multicast address that you no longer want to use.

DESCRIPTION

Each time a client of either the Ethernet or FDDI hardware device driver calls the `EAddMulti` function, the driver increments a counter for the multicast address specified by the `eMultiAddr` parameter. Each time a client of either the Ethernet or FDDI hardware device driver calls the `EDelMulti` function, the driver decrements the counter for the address specified by the `eMultiAddr` parameter.

As long as the count for a multicast address is equal to or greater than 1, the hardware device driver accepts packets directed to that multicast address. When the count for an address equals 0, the driver removes that address from the list of multicast addresses that it accepts. For token ring, the same process applies to functional addresses.

SPECIAL CONSIDERATIONS

Because more than one client of the .ENET driver might be using a particular multicast address, you should call the `EDelMulti` function only once for each time you called the `EAddMulti` function.

TOKEN RING AND FDDI CONSIDERATIONS

If your application added a multicast address for FDDI, you use this function to delete the address when you no longer need it. If your application added a functional address for token ring, use this function to delete the address when you no longer need it. Functional addresses are the token ring equivalent of Ethernet and FDDI multicast addresses. Be careful not to use the broadcast address as either a multicast or a functional address. (For information on all three types of addresses, see *Inside AppleTalk,* second edition.)

ASSEMBLY-LANGUAGE INFORMATION

To execute the `EDelMulti` function from assembly language, call the `_Control` trap macro with a value of `ENetDelMulti` in the `csCode` field of the parameter block. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field.

RESULT CODES

`noErr`	0	No error
`eMultiErr`	–91	Address not found

Summary of Ethernet, Token Ring, and FDDI

Pascal Summary

Constants

```
CONST
{.ENET, .TOKN, and .FDDI driver values}
   catNetwork              = 4;          {spCategory for Ethernet NB card}
   typeEtherNet            = 1;          {spCType for Ethernet NB card}
   typeTokenRing           = 2;          {spCType for token ring NB card}
   typeFDDI                = 11;         {spCType for FDDI NB card}

{.ENET driver routine selectors}
   ENetSetGeneral          = 253;        {set to general transmission mode}
   ENetGetInfo             = 252;        {get info}
   ENetRdCancel            = 251;        {cancel read}
   ENetRead                = 250;        {read}
   ENetWrite               = 249;        {write}
   ENetDetachPH            = 248;        {detach protocol handler}
   ENetAttachPH            = 247;        {attach protocol handler}
   ENetAddMulti            = 246;        {add a multicast address}
   ENetDelMulti            = 245;        {delete a multicast address}
```

Data Structures

```
TYPE EParamBlock =
      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 Integer OF
         ENetWrite, ENetAttachPH, ENetDetachPH, ENetRead, ENetRdCancel,
            ENetGetInfo, ENetSetGeneral:
            (
            eProtType:     Integer;       {Ethernet protocol type}
            ePointer:      Ptr;           {pointer; use depends on }
                                          { function}
            eBuffSize:     Integer;       {buffer size}
            eDataSize:     Integer;       {number of bytes read}
            );

         ENetAddMulti,ENetDelMulti:
            (
            eMultiAddr:    ARRAY[0..5] OF Char; {multicast address}
            )
         END;

EParamBlkPtr = ^EParamBlock;
```

Routines

Attaching and Detaching an Ethernet Protocol Handler

```
FUNCTION EAttachPH          (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
FUNCTION EDetachPH          (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

Writing and Reading Ethernet Packets

```
FUNCTION EWrite             (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
FUNCTION ERead              (thePBptr: EParamBlkPtr; async: Boolean):OSErr;
FUNCTION ERdCancel          (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

Obtaining Information About the Ethernet Driver and Switching Its Mode

```
FUNCTION EGetInfo           (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
FUNCTION ESetGeneral        (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

Adding and Removing Ethernet Multicast Addresses

```
FUNCTION EAddMulti          (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
FUNCTION EDelMulti          (thePBptr: EParamBlkPtr; async: Boolean): OSErr;
```

C Summary

Constants

```
enum {
   ENetSetGeneral    = 253,         /*set "general" mode*/
   ENetGetInfo       = 252,         /*get info*/
   ENetRdCancel      = 251,         /*cancel read*/
   ENetRead          = 250,         /*read*/
   ENetWrite         = 249,         /*write*/
   ENetDetachPH      = 248,         /*detach protocol handler*/
   ENetAttachPH      = 247,         /*attach protocol handler*/
   ENetAddMulti      = 246,         /*add a multicast address*/
   ENetDelMulti      = 245,         /*delete a multicast address*/
};
```

Data Types

```
#define EParamHeader \
   QElem       *qLink;             /*reserved*/\
   short       qType;              /*reserved*/\
   short       ioTrap;             /*reserved*/\
   Ptr         ioCmdAddr;          /*reserved*/\
   ProcPtr     ioCompletion;       /*completion routine*/\
   OSErr       ioResult;           /*result code*/\
   StringPtr   ioNamePtr;          /*reserved*/\
   short       ioVRefNum;          /*reserved*/\
   short       ioRefNum;           /*driver reference number*/\
   short       csCode;             /*call command code*/

struct EParamMisc1 {
   EParamHeader                    /*general EParams*/
   short       eProtType;          /*Ethernet protocol type*/
   Ptr         ePointer;
   short       eBuffSize;          /*buffer size*/
   short       eDataSize;          /*number of bytes read*/
};
```

Note
The C interface file contains the following structure type definition, which is incorrect. A corrected version follows it. ◆

```
typedef struct EParamMisc1 EParamMisc1;

struct EParamMisc2 {
   EParamMisc1 EParms1;
   char  eMultiAddr[6];            /*multicast address*/
};
```

Note
The following structure type definition is a correction to the preceding structure that may exist in the interface file. You should declare the following struct in your application instead of relying on the interface file. ◆

```
typedef struct {
      EParamHeader
      char eMultiAddr[5];          /*multicast address*/
}EParamMisc2;

typedef struct EParamMisc2 EParamMisc2;

union EParamBlock {
   EParamMisc1 EParms1;
   EParamMisc2 EParms2;
};

typedef union EParamBlock EParamBlock;

typedef EParamBlock *EParamBlkPtr;
```

Routines

Attaching and Detaching an Ethernet Protocol Handler

```
pascal OSErr EAttachPH      (EParamBlkPtr thePBptr, Boolean async);
pascal OSErr EDetachPH      (EParamBlkPtr thePBptr, Boolean async);
```

Writing and Reading Ethernet Packets

```
pascal OSErr EWrite         (EParamBlkPtr thePBptr, Boolean async);
pascal OSErr ERead          (EParamBlkPtr thePBptr, Boolean async);
pascal OSErr ERdCancel      (EParamBlkPtr thePBptr, Boolean async);
```

Obtaining Information About the Ethernet Driver and Switching Its Mode

```
pascal OSErr EGetInfo       (EParamBlkPtr thePBptr, Boolean async);
pascal OSErr ESetGeneral    (EParamBlkPtr thePBptr, Boolean async);
```

Adding and Removing Ethernet Multicast Addresses

```
pascal OSErr EAddMulti      (EParamBlkPtr thePBptr, Boolean async);
pascal OSErr EDelMulti      (EParamBlkPtr thePBptr, Boolean async);
```

Assembly-Language Summary

Constants

```
ENetSetGeneral    EQU     253       ;set to general transmission mode
ENetGetInfo       EQU     252       ;get info
ENetRdCancel      EQU     251       ;cancel read
ENetRead          EQU     250       ;read
ENetWrite         EQU     249       ;write
ENetDetachPH      EQU     248       ;detach protocol handler
ENetAttachPH      EQU     247       ;attach protocol handler
ENetAddMulti      EQU     246       ;add a multicast address
ENetDelMulti      EQU     245       ;delete a multicast address
```

Data Structures

EParamBlock Parameter Block

16	`ioResult`	word	result code
26	`csCode`	word	routine selected
28	`eMultiAddr`	6 bytes	multicast address
28	`eProtType`	word	Ethernet protocol type
30	`ePointer`	long	pointer
34	`eBuffSize`	word	size of buffer
36	`eDataSize`	word	number of bytes read

Result Codes

`noErr`	0	No error
`eMultiErr`	–91	Address not found
`eLenErr`	–92	Packet too large or first entry of the write-data structure did not contain the full 14-byte header
`LAPProtErr`	–94	No protocol handler is attached
`excessCollsns`	–95	Hardware error
`memFullErr`	–108	Insufficient memory in heap
`cbNotFound`	–1102	`ERead` not active
`reqAborted`	–1105	`ERdCancel` or `EDetachPH` function called
`buf2SmallErr`	–3101	Packet too large for buffer; partial data returned