Multinode Architecture

This chapter describes how you can use AppleTalk's multinode architecture to acquire one or more node IDs, called ***multinodes,*** in addition to the standard user node ID. ***Multinode architecture*** is an AppleTalk feature that is provided to meet the needs of special-purpose applications that receive and process AppleTalk packets in a custom manner instead of passing them directly on to a higher-level AppleTalk protocol for processing. A multinode ID allows the system that is running your application to appear as multiple nodes on the network. The prime example of a multinode application is Apple Remote Access (ARA).

A multinode ID is distinct from the user node ID. AppleTalk separates packets addressed to a multinode from those addressed to the user node sockets on the same machine, and it passes the multinode packets on to a receive routine that you must supply for the multinode.

Multinode architecture is implemented in the .MPP driver and exists at the same level of the AppleTalk protocol stack as does the Datagram Delivery Protocol (DDP), but unlike DDP, multinode does not use DDP sockets, nor is it connected to the AppleTalk protocol stack above the data-link level.

This chapter describes the fundamental tasks that you perform to

- add a multinode for your application's use
- write a required routine that receives packets addressed to the multinode
- prepare and send data from the multinode
- remove a multinode when you are finished with it

Because multinode is not connected to the AppleTalk protocol stack above the data-link level, if you want your multinode application to be compatible with AppleTalk, you must implement the higher-level AppleTalk protocols. Multinode also requires that you code a receive routine in assembly language. For these reasons, you should consider using multinode only if your application requires that you process AppleTalk packets in a custom manner. You do not need to use the multinode architecture for other application requirements.

The receive routine that you must provide to handle packets addressed to your multinode ID is similar to the DDP socket-listener code that an application must include to receive packets addressed to its DDP socket. The chapter "Datagram Delivery Protocol (DDP)" in this book describes how to write a socket listener, which provides useful background information on how to write a multinode receive routine.

At the data-link level, multinode architecture relies on the AppleTalk connection file of type `'adev'` that is implemented for a particular link type. For more information about AppleTalk connection files, see the *Macintosh AppleTalk Connections Programmer's Guide*.

For information describing how to implement the higher-level AppleTalk protocols, see *Inside AppleTalk*, second edition.

About Multinode Architecture

AppleTalk multinode architecture lets you acquire multiple node addresses for a single machine, allowing that machine to act and appear as several nodes on a network. You can think of a multinode as a virtual node and the user node as the physical node. A single machine or physical node can have associated with it one or more multinodes. You can obtain a multinode ID after a node that is running your application connects to the AppleTalk network and AppleTalk assigns the standard user node ID to that system. The use of multinode addresses does not affect the functions of the standard user node address, which uniquely identifies the physical node on the network and forms part of the internet socket address of a DDP socket-client application.

Multinode architecture communicates similarly to DDP in that you send data from a multinode as discrete packets, with each packet carrying the full addressing information required to deliver the data to its destination.

Multinode architecture is a client of the data-link layer and all of the supported data-link types. It is connected to the AppleTalk protocol stack from the data-link layer down through the hardware. It is not connected to the AppleTalk protocols above it, and there are no hooks that a multinode application can use to pass a packet up through the AppleTalk protocol stack for processing by a higher-level protocol.

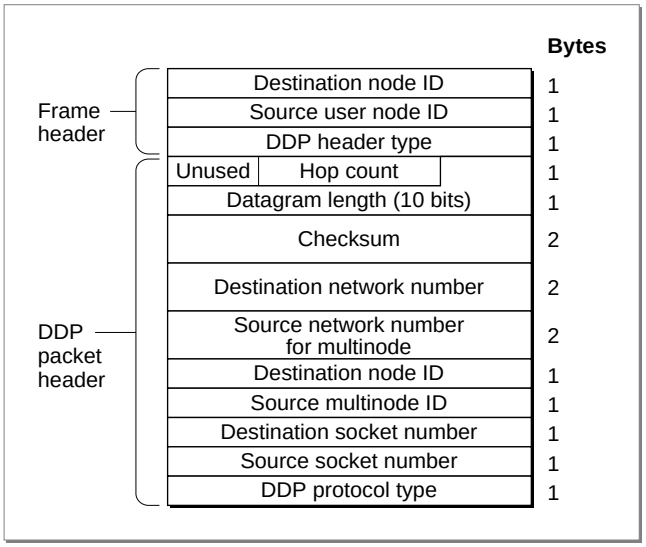
Therefore, a multinode application that receives DDP packets for higher-level AppleTalk protocols must process these packets itself, in its own way. For example, if a multinode application receives an AppleTalk AEP Echoer request packet, it must determine how to handle the request packet, that is, whether or not to respond to the packet as the AppleTalk Echo Protocol (AEP) implementation does. (For more information on AEP, see the discussion in the chapter "Datagram Delivery Protocol [DDP]" in this book and the AEP protocol specification in *Inside AppleTalk*, second edition.)

After a packet is delivered to the node, the .MPP driver checks the DDP packet header and passes packets addressed to a user node socket on to the appropriate socket listener, while passing packets addressed to a multinode on to the receive routine that you provide as part of your multinode application. Your receive routine must receive both packets addressed to the multinode and broadcast packets. A receive routine is similar to a socket listener. You must code the receive routine in assembly language because the .MPP driver passes values to your routine in registers when it calls the routine.

Multinode architecture does not provide for the establishing of sessions—that is, the ability to set up a connection and send streams of data over it, nor does it include support for error recovery. If you want these features, you need to provide them in your multinode application.

AppleTalk delivers all packets to the physical node based on the user node ID assigned to the node, which is carried in the frame header as the destination node ID. Multinode architecture always uses a long DDP packet header; Figure 12-1 shows the structure of the long DDP packet header. It also shows the frame header.

Figure 12-1 The long DDP packet header used for multinode



When you send a packet from a multinode:

- The frame header always contains the source user node ID, which identifies the physical node on the network from which the packet was transmitted.
- The DDP packet header always contains the source multinode ID, which identifies the virtual multinode from which you are sending the packet.

A packet is always transmitted from the physical node's network hardware, and the frame header contains the user node ID of the physical node that transmitted the packet. Your multinode application uses a multinode ID, which you can think of as a virtual node from which you are sending data. The DDP header identifies this multinode. Your application sends data, but the networking hardware and its device driver actually transmit the packet containing the data across the network to its destination.

A single networked machine may have associated with it one or more multinode IDs. Packets sent from several multinode applications running on the same machine include different source multinode IDs, but because they are all transmitted from the same physical node, the packets all have the same source user node ID.

Because the source multinode ID is associated with the application that sent the packet and the source user node ID is associated with the machine that transmitted the packet, the source user node ID in the frame header and the source multinode ID in the DDP packet header are always different values.

Note
Even if the destination node of a packet is on the same LocalTalk network as the source user node, a packet sent from a multinode always contains a long DDP header to allow for the inclusion of the two separate source node IDs: the user node ID and the multinode ID. ◆

To acquire a multinode, you call the `AddNode` routine. You can obtain only one multinode at a time. The number of multinodes that a single machine can support is limited by the maximum number of multinodes supported by the underlying AppleTalk connection file of type `'adev'` for the data link that is being used:

- For LocalTalk, the maximum is 254 node addresses ($0 and $FF are not valid addresses).
- For EtherTalk, TokenTalk, or FDDITalk, the maximum is 253 node addresses ($0, $FF, and $FE are not valid addresses).

Because the multinode is considered another unique node ID, the number of multinodes that can be acquired is further limited by the number of nodes already active on the network.

As an example of one use of multinodes, consider how a multinode application that includes server and client components might handle a broadcast NBP lookup packet. The following events occur on the user node that runs the client component of the multinode application:

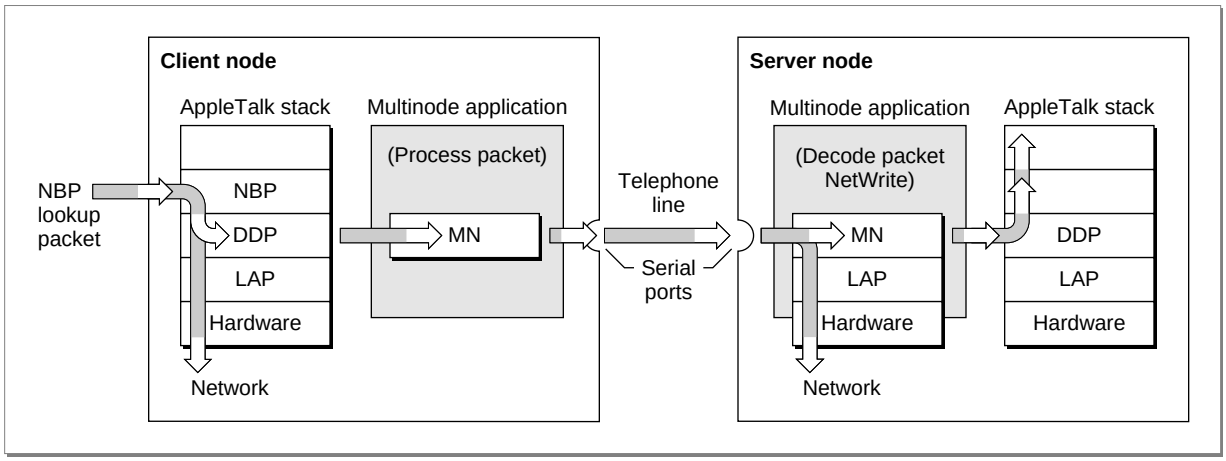
1. A DDP socket-client application on the user node calls an NBP function that generates a broadcast NBP lookup packet.
2. The .MPP driver sends the packet out to the network. Because it is a broadcast packet, the .MPP driver also sends the NBP lookup packet to the multinode on the same machine.
3. The multinode client application's receive routine receives the packet.
4. The multinode client application processes the packet's contents and repackages them in its own multinode packet, which it sends out through the serial port over the modem and telephone line to the multinode application on the server node.

The following events occur on the node that is running the server component of the multinode application.

1. The server multinode application receives the multinode packet through the system's serial port.
2. This application uses the `NetWrite` routine to decode the multinode packet and uses the packet contents as the data for a DDP packet. It builds the required data structure to contain the data for a standard DDP packet.
3. The server multinode application then sends the broadcast packet down through the AppleTalk protocol stack from the link-access layer, through the hardware, and out to the network for a response. It also sends the packet to the user node on the same machine.

Figure 12-2 illustrates this process.

Figure 12-2 How a server-client multinode application might send a broadcast NBP lookup packet



The primary use of the multinode architecture for an application is to provide router-like services as part of the application. One of the advantages of multinode is that your application receives all Name-Binding Protocol (NBP) request packets because they are broadcast packets. In fact, the first packets that your application is likely to receive are NBP lookup packets. These include NBP register requests that generate an NBP lookup request if the sender specified that NBP should verify the uniqueness of the entity name to be registered. (For an explanation of NBP and its components, see the chapter "Name-Binding Protocol [NBP]" in this book.)

How you handle the NBP lookup packets is application-specific. However, if you want your application to be visible throughout the network, you need to meet certain AppleTalk compatibility requirements. In this case, your application needs to implement the NBP protocol. You can implement your own NBP names table for the multinode to determine if your application handles the services requested in the lookup packet. For example, your application can check to determine if an NBP lookup packet's entity name object and type fields match the object and type fields of any of the entity name entries in your NBP names table. Any response that you return to the requester must conform to the AppleTalk packet format. You may also want to implement the AppleTalk Echo Protocol (AEP), and in this case, too, any responses that you return to the sender must meet the specifications for an AEP AppleTalk packet. (For a description of AEP, see the chapter "Datagram Delivery Protocol [DDP]" in this book.) *Inside AppleTalk*, second edition, describes how to implement NBP and AEP.

Using Multinode Architecture

This section describes how to

- acquire a multinode (`AddNode`)
- receive data addressed to the multinode
- prepare to send data and then send it from the multinode (`NetWrite`)
- remove a multinode when you are finished with it (`RemoveNode`)

It also mentions the cable-range-change AppleTalk transition event that you must handle and directs you to the chapter "Link-Access Protocol (LAP) Manager" for information describing what you must do.

The routines that you use to add and remove a multinode and send data from your multinode application are not defined in the MPW interface files. To use these routines from a high-level language, you must call the Device Manager directly and specify the routine's `csCode` in the parameter block. For the `AddNode` routine, you must issue the function as an immediate control call and define a function for this purpose. (For an example of how to do this, see Listing 12-1 on page 12-9.) For the `NetWrite` and `RemoveNode` routines, you call the Device Manager's `PBControl` function. (For information about how to do this, see "Routines" beginning on page 12-20.)

Note
AppleTalk version 57 or later must be installed on the system that is running your application if you use the multinode feature. AppleTalk version 57 is compatible with system software version 6.0.5 and later. You should include AppleTalk version 57 with any product that uses multinodes. Contact Apple's Software Licensing department for information on licensing AppleTalk. ◆

Acquiring and Removing Multinodes

You can add an AppleTalk multinode once the physical node that runs your application has connected to the AppleTalk network and AppleTalk has assigned to it a user node ID. After you are finished using the multinode, your application must remove it. This section describes how to do these tasks.

To acquire a multinode address, perform the following steps:

1. Use the Device Manager's `OpenDriver` function to open the .MPP driver.
   - The .MPP driver must be opened before you call the multinode routines. The `OpenDriver` function call returns the .MPP driver's reference number.
   - Save the returned value because you must supply this reference number as an input parameter in the `ioRefNum` field of the multinode parameter block when you call the multinode routines.

2. Create a receive routine to receive broadcast messages and packets addressed to your multinode. See "Receiving Packets Addressed to Your Multinode" beginning on page 12-10 for details.
   - You pass the address of the receive routine to the .MPP driver when you call the `AddNode` routine to acquire a multinode.
   - When the .MPP driver receives a packet addressed to your multinode or a broadcast message, it calls your receive routine for that multinode to handle the packet reception.
3. Allocate storage and set parameter block fields as needed.
   - Define a multinode parameter block of type `MNParamBlock`. Allocate storage for a multinode parameter block that includes the fields required for the `AddNode` routine. See "The Multinode Parameter Block" on page 12-19.
   - You must set the `csCode` parameter block field to the numeric value of 262 for the `AddNode` routine. For the other required parameter block fields, see "AddNode" beginning on page 12-22.
4. Call the `AddNode` routine once for each multinode that you need.
   - You can acquire only one multinode through each request. You can request a specific multinode address, and if that multinode is available, the .MPP driver will assign it to you. Otherwise, the .MPP driver will return a multinode address that it selects randomly.
   - Because the `AddNode` routine is not defined in the MPW interface files, you must call the Device Manager directly and execute the `AddNode` routine as an immediate synchronous control call.

From assembly language, you can directly make an `immed _Control` trap macro call. To issue the `AddNode` routine as an immediate synchronous control call from a high-level language such as Pascal or C, you must define a function as part of your application. Listing 12-1 shows how to do this in the Pascal language.

Listing 12-1 Defining a Pascal function that makes an immediate `AddNode` call

```
FUNCTION PBControlImmedSync(paramBlock: ParmBlkPtr): OSErr;
   INLINE $205F,$A204,$3E80;

FUNCTION AddNode(thePBptr: MNParmBlkPtr): OSErr;
CONST
   tryAddNodeAgainErr   =  -1021;
VAR
   err: OSErr;

BEGIN
   thePBptr^.csCode  := 262; {addNode}
   thePBptr^.ioRefNum := mppUnitNum;
{If the call returns tryAddNodeAgainErr, make the call repeatedly
until it no longer returns this error.}
```

```
   REPEAT
      err   := PBControlImmedSync(ParmBlkPtr(thePBptr));
      UNTIL (err <> tryAddNodeAgainErr);
      AddNode := err;
END;
```

You must issue the `AddNode` call synchronously because you need to call `AddNode` repeatedly if the call returns an error of –1021, which indicates that the .MPP driver could not satisfy the `AddNode` request and that you should try the request again immediately.

The .MPP driver internally associates the address of your receive routine with the multinode address that it returns to you. See "AddNode" beginning on page 12-22 for a complete description of this routine and the parameters that you must pass it.

When you are finished using the multinode, you call the `RemoveNode` routine to remove the multinode.

1. Allocate nonrelocatable memory for a multinode parameter block that includes the fields required for the `RemoveNode` routine. See "The Multinode Parameter Block" beginning on page 12-19. The multinode parameter block belongs to the .MPP driver for the life of the `RemoveNode` call.
2. You issue the `RemoveNode` routine as a Device Manager's `PBControl` call. See "RemoveNode" beginning on page 12-24 for details on this routine and the parameters it requires. You must specify the `csCode` numeric value 263 for the `RemoveNode` routine.

Handling an AppleTalk Cable-Range-Change Transition Event

A cable range is a range of network numbers beginning with the lowest network number and ending with the highest network number defined by a seed router for a network. All node addresses, including multinode addresses, that a system on a network acquires must have a network number within the defined cable range.

An AppleTalk cable-range-change transition event occurs when the current cable range for a network changes. Your multinode application needs to be able to receive notification of a cable-range-change transition and respond to that event by checking the new cable range to determine if all the multinode IDs that the application acquired before the transition event occurred are still valid. If you discover multinode IDs that are no longer valid, you must remove them with the `RemoveNode` function. You can obtain new multinodes to replace them with the `AddNode` function.

Receiving Packets Addressed to Your Multinode

Your application must provide a routine that receives packets addressed to the multinode and broadcast packets. Because the .MPP driver passes values to your multinode receive routine in registers when it calls the routine, you must code the receive routine in assembly language.

You pass the address of your receive routine to the .MPP driver when you call the `AddNode` routine to open a multinode. The .MPP driver internally associates your receive routine with the multinode address that it assigns, and it calls your receive routine to handle a packet addressed to the multinode or a broadcast packet.

If your application acquires more than one multinode, you can use the same receive routine for each of these multinodes. If you use the same receive routine to receive and process packets for more than one multinode, the .MPP driver will call that receive routine only once for each broadcast packet that it receives.

A multinode receive routine is similar in concept to a socket listener that receives packets addressed to a specific socket. The chapter "Datagram Delivery Protocol (DDP)" in this book includes a sample socket listener. To create a receive routine, perform the following steps:

1. Allocate a buffer to hold the data that you expect to receive.
   - The maximum amount of data in a DDP packet is 586 bytes. All packets addressed to multinodes use a long header, which is 13 bytes long. If your receive routine places the packet header as well as the data portion in the buffer, make the buffer large enough to hold both parts of the packet contents.
   - If you use the same receive routine to receive and process packets for more than one multinode, you should provide a separate buffer to store the data for each multinode. You can define a single buffer for each multinode to hold the contents of both the header and data portions of a packet, or you can define a pair of buffers for each multinode to separate the packet's contents.
2. Determine the number of bytes that have already been read into the .MPP driver's internal buffer, called the RHA.
   - To do this, subtract the beginning address of the *read-header area (RHA)* from the value in register A3, which points past the last byte read into the RHA. To locate the offset at the beginning of the RHA, you can use the `toRHA` equate.

   When a frame that contains either a DDP packet that is addressed to your multinode or a broadcast packet is delivered to the node that is running your multinode application, the node's CPU is interrupted and the .MPP driver's interrupt handler gets control to service the interrupt. As the frame's first 3 bytes are read into a FIFO buffer, the .MPP driver's interrupt handler moves these bytes into the RHA.
3. Use the `ReadPacket` and `ReadRest` routines to read the rest of the incoming data that constitutes the packet.

   How you handle a packet after you read it is particular to your application. For example, if your application implements NBP, you can check the packet's entity name object and type fields against entries in your names table to determine whether to process the packet and respond to the sender. If you respond, the packet you send must adhere to the structure of a standard AppleTalk packet. (See *Inside AppleTalk,* second edition, for the AppleTalk packet structure.) For a brief description of how ARA uses multinode, see the discussion on page 12-6.
   - You can call the `ReadPacket` routine as many times as you like to read the data piece by piece into one or more data buffers that you have defined, 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 for which you allocated memory in the A3 register. You must also place the number of bytes you want to read in the D3 register. You must not request more bytes than remain in the data packet.
- After you have called the `ReadRest` routine, you can use registers A0 through A3 and D0 through D3 for your own use, but you must preserve all other registers. You cannot depend on having access to your application's global variables.

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

If LocalTalk is the data link that is being used, your receive routine has less than 95 microseconds (best case) to read more data with a `ReadPacket` or `ReadRest` routine. If you need more time, you can read another 3 bytes into the RHA, which will allow you an additional 95 microseconds. Note that the RHA may only have 8 bytes still available. ◆

4. If the packet header contains a checksum, you can calculate a checksum for both the header and data portions of the packet and then verify the sum of these two values against the value in the `checksum` field of the packet header. If the checksum you calculate does not match the one in the header, the data has been corrupted in some way. (Figure 12-1 on page 12-5 shows the DDP packet header, including the checksum field.)

   The chapter "Datagram Delivery Protocol (DDP)" in this book contains a sample checksum routine to be used for a socket listener; this routine is equally applicable to a multinode receive routine.

Calling ReadPacket to Read in the Packet Contents

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 Size in of bytes to be read; must be nonzero

Registers on exit from the `ReadPacket` routine

A0 Unchanged

A1 Unchanged

A2 Unchanged

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

A4 Unchanged

D0 Changed

D1 Number of bytes left to be read

D2 Unchanged

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

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

Calling ReadRest to Complete Reading in the Packet Contents

Call the `ReadRest` routine to read the last portion of the data packet, or call it after you have read all the data with `ReadPacket` routines and before you do any other processing or terminate execution. After you call `ReadRest`, you must check the zero (`z`) flag in the status register for errors.

After you call the `ReadRest` routine, you must terminate execution of your receive routine with an RTS instruction whether or not the `ReadRest` routine returns an error.

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

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

To call the `ReadRest` routine, execute a JSR instruction to an address 2 bytes past the address in the A4 register:

```
JSR 2(A4)
```

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: number of bytes left to be read

D2 Unchanged

D3 Equals 0 if the requested number of bytes were read, is less than 0 if the packet data was too large to fit in the buffer and the data was truncated, and is greater than 0 to indicate the number of bytes that were not read

For more information on how your receive routine can use the registers, see the discussion of the socket listener routine in the chapter "Datagram Delivery Protocol (DDP)" in this book.

Sending Packets Using a Multinode

You can use a multinode to send packets that contain data that you have already received; in this case you forward the data from the multinode using the `NetWrite` call. You can also use the multinode to send original data using the `NetWrite` call. In both cases, you must use a structure called the ***write-data structure*** to indicate to the .MPP driver where the DDP packet header portion and the data portion to be sent are stored. Why you send data is particular to your application. For example, if your application implements AEP, it would send an Echo Reply packet in response to the Echo Request packet that the application receives. For a brief description of using multinode, see the discussion on page 12-6.

To send data from the multinode, you perform the following steps:

1. Create a write-data structure, as described in the next section, "Preparing a Write-Data Structure."
2. Allocate nonrelocatable memory for a multinode parameter block that includes the fields required for the `NetWrite` routine. See "The Multinode Parameter Block" beginning on page 12-19. The multinode parameter block belongs to the .MPP driver for the life of the `NetWrite` call.
3. Call the `NetWrite` routine to send the data. You issue the `NetWrite` routine as a Device Manager's `PBControl` call. See "NetWrite" beginning on page 12-25 for details on this routine and the parameters it requires.
   - Set the parameter block field values belonging to the `NetWrite` call, including the checksum flag (`checkSumFlag`) parameter. See "Using a Checksum" on page 12-16.
   - You must set the `csCode` parameter block field to the numeric value of 261 for the `NetWrite` routine.

Preparing a Write-Data Structure

The .MPP driver uses a write-data structure that you create to locate the header and data portions of the packet to be transmitted. When you call the `NetWrite` routine to send data from a multinode, you pass it a pointer to the write-data structure that you have already prepared. A write-data structure contains a series of pairs of length words and pointers, and each pair indicates the length and location of a portion of the data. The first pair must indicate the DDP header of the packet to be transmitted. It ends with a 0 word.

The .MPP driver constructs the packet to be transmitted, building the packet contents from the header and data information that you provide.

The write-data structure that you use for a multinode is similar to the write-data structure that you use to send a packet from a DDP socket except that for a multinode write-data structure, you must also include the source network number and the source multinode ID. This is because the source user node ID of the physical node, which is carried in the frame header, is different from the source multinode ID, which is carried in

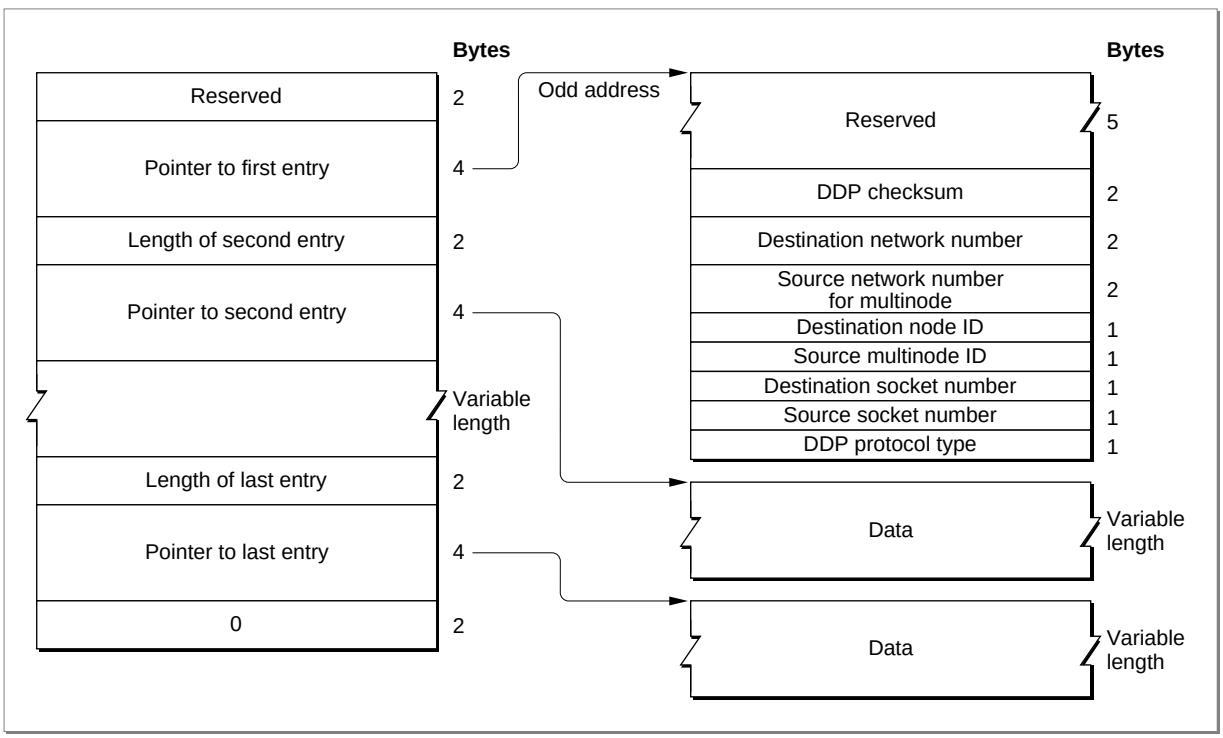
the DDP packet header. The source address information that you provide identifies the multinode from which you are sending the data. The multinode write-data structure also contains a checksum field that you can set to 0 if you do not want a checksum calculated for this packet. Figure 12-3 shows the write-data structure; it also shows how you must define the header information in the storage that you allocate for it.

You create a write-data structure in one of three different forms:

- You can provide a single length-pointer pair that identifies one storage block that contains both the header and data information. In this case, the header information must come first, and it must begin at an odd address.
- You can use two length-pointer pairs, one for the header portion and one for the data portion.
- You could also use more than two length-pointer pairs, one for the header, and one for each separate block of data.

In many cases, the header and data components of a packet are not stored contiguously, which requires that the write-data structure contain at least two length-pointer pairs. Typically, the data portion is stored as a single block. However, some implementations send blocks of data that are stored separately as parts of the same datagram; if the complete data portion is stored as several separate blocks, then the write-data structure needs to contain a length-pointer pair for each block of data.

Figure 12-3 The write-data structure for a multinode



Note
The header block that the write-data structure points to consists of 16 bytes. The first pointer in the write-data structure must point to an odd address, so if you create the write-data structure in Pascal, the first byte is not used. ◆

For the header, you must fill in the following:

- the destination network number
- the source network number of the multinode
- the destination node ID
- the source multinode ID
- the destination socket number
- the source socket number (if you are forwarding from the multinode a DDP packet that contains an existing value for the source socket number, you can pass that value on in this field)
- DDP protocol type (DDP protocol types 1 through 15 are reserved for use by Apple)

Note
A multinode is not associated with a DDP socket. If the source socket field contains a value, it must adhere to the conventions that the AppleTalk DDP protocol specification describes for the use of sockets. For example, this field must not specify socket number 0 ($00); rather the value should be constrained to socket number values belonging to the user-defined range stated in the DDP protocol specification; see *Inside AppleTalk*, second edition, for this information. ◆

Using a Checksum

The long DDP packet header that you create for a multinode can include a checksum value that is used to verify that the packet data has not been corrupted by memory or data bus errors within routers on the internet. When you call the `NetWrite` routine to send data from a multinode, you specify a value for the `checkSumFlag` parameter of the multinode parameter block. You use the `checkSumFlag` parameter differently to send data from a multinode than how you would use it to send data from a DDP socket, even though in both cases the flag's value controls the use of the long DDP packet header's `checksum` field.

Any application that uses a multinode can receive packets through that multinode. The application can then repackage and forward the packet through the serial port and modem to its multinode-application counterpart on a remote system. The multinode application at the remote end can then decode the package and send the packet on through a `NetWrite` call to a node on the network or a user-node process on the same machine. An existing packet that is to be forwarded could already contain a checksum

value. When you issue the `NetWrite` call, you can preserve that checksum value and pass it on as part of the header in the packet. You use the `checkSumFlag` parameter of the `NetWrite` routine for this purpose.

- If you do not want the current value in the packet header's checksum field to be altered, you set `checkSumFlag` to 0, and the existing checksum value in the DDP header will not be changed. (If a checksum has already been calculated, it will be passed along unmodified.)
- If you want the checksum for the datagram to be calculated and placed in the DDP packet header's checksum field before the .MPP driver transmits the packet, set `checkSumFlag` to a nonzero number.

Note that if you want to send a packet that does not include a checksum, you must hardcode the value by setting to 0 the checksum field of the data structure that contains the packet header that you point to from the write-data structure.

How the Apple Remote Access program uses the checksum flag

The Apple Remote Access (ARA) program is an example of an application that sets the `checkSumFlag` flag to 0 in order to preserve a packet's original checksum value. The ARA client multinode can receive a DDP packet addressed to that multinode or a broadcast packet, such as an NBP lookup packet. In either case, the packet is a standard DDP packet that could contain a checksum value. The client ARA software passes the packet on to the ARA software on the server through the serial port and modem. The ARA software on the server node sets `checkSumFlag` to 0 when it calls the `NetWrite` routine to send the packet down from the multinode through the AppleTalk stack and out to a node on the network. ◆

Multinode Architecture Reference

This section describes the data structures and routines that are specific to the multinode architecture.

The "Data Structures" section shows the Pascal data structures for the write-data structure, the address block record, and the multinode parameter block to the .MPP driver.

The "Routines" section describes the routines that you use to add a multinode address, remove a multinode address, and send data from the multinode to be transmitted over the network. Unlike most of the routines comprising the protocol implementations described in this book, the multinode routines are not defined in the MPW interface files. To call these routines from a high-level language, you must use the Device Manager's interface. The "Routines" section describes how to do this.

Data Structures

This section describes the data structures that you use to provide information to the multinode architecture implementation in the .MPP driver.

- You use the write-data structure to pass information to the `NetWrite` routine that identifies the length and location of the header and data portions of a packet to be sent from the multinode.
- You use the address block record to pass to the `AddNode` routine the address of the multinode that you wish to acquire and to receive from the routine the actual multinode address that the .MPP driver assigns.
- You use the multinode parameter block to pass and receive the input and output parameters for each multinode call.

The Write-Data Structure

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, including the header information, that constitutes the packet to be sent over the network.

You create a write-data structure, then pass its pointer to the `NetWrite` routine to send a packet from a multinode.

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

Field descriptions

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

`entryPtr` A pointer to the data that is part of the packet to be sent using the `NetWrite` routine. The data storage area pointed to can contain the header information, the data to be transmitted, or both.

The Address Block Record

The address block record defines a data structure of `AddrBlock` type. You use this record type for

- the `reqNodeAddr` field value of the multinode parameter block to specify the preferred network number and multinode ID of the multinode that you wish to acquire when you execute the `AddNode` routine
- the `actNodeAddr` parameter block field for the `AddNode` routine for the .MPP driver to return the multinode address that it assigns to you
- the `nodeAddr` parameter block field for the `RemoveNode` routine to specify the address of the multinode to be removed

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

Field descriptions

`aNet`	The number of the desired network to which the multinode node that you are requesting or assigned belongs.
`aNode`	The node ID of the multinode that you request or that MPP assigns.
`aSocket`	The value of this field should always be 0.

The Multinode Parameter Block

The multinode routines that you use to add and remove a node and send a packet from a multinode require a pointer to a multinode parameter block. The multinode parameter block holds all of the input and output values associated with the routine. The multinode parameter block is a variant record parameter block, defined by the `MNParamBlock` data type.

IMPORTANT

For the multinode parameter block, you must define the `MNParamBlock` type in your application because it is not included in the MPW interface files. ▲

This section defines the fields that are common to the three multinode routines that use the multinode parameter block. It does not define reserved fields, which are used either internally by the .MPP driver or not at all. The fields that are used for specific routines only are defined in the description of the routines to which they apply.

```
TYPE
MNParmType =   (AddNodeParm,RemoveNodeParm);
MNParamBlock   =
PACKED RECORD
      qLink:           QElemPtr;        {reserved}
      qType:           Integer;         {reserved}
      ioTrap:          Integer;         {reserved}
      ioCmdAddr:       Ptr;             {reserved}
      ioCompletion:    ProcPtr;         {completion routine}
      ioResult:        OSErr;           {result code}
      ioNamePtr:       StringPtr;       {reserved}
      ioVRefNum:       Integer;         {reserved}
      ioRefNum:        Integer;         {driver reference number}
      csCode:          Integer;         {command code}
      filler1:         Byte;
```

```
checkSumFlag:     Byte;             {perform checksum on datagram}
   wdsPointer:    Ptr;              {pointer to write-data structure}
   filler2:       Integer;
   CASE MNParmType of
      AddNodeParm:
         (reqNodeAddr:  AddrBlock;  {preferred address requested}
         actNodeAddr:   AddrBlock;  {actual node address acquired}
         recvRoutine:   ProcPtr;    {address of packet receive routine}
         reqCableLo:    Integer;    {preferred network range for the }
         reqCableHi:    Integer;    { node being acquired}
         reserved: PACKED ARRAY[1..70] OF Byte);
      RemoveNodeParm:
         (nodeAddr: AddrBlock);     {node address to be deleted}
   END;
```

Field descriptions

`ioResult`	The result of the function. When you execute the function asynchronously, the function sets this field to 1 and returns a function result of `noErr` as soon as the function begins execution. When the function completes execution, it sets the `ioResult` field to the actual result code.
`ioRefNum`	The .MPP driver reference number. You must fill in this value.
`csCode`	The command code of the multinode command to be executed. You must fill in a numeric value for this field.

Routines

This section describes the multinode routines that you use to

- acquire a multinode address
- remove a multinode address once you are finished with it
- send packets from a specific multinode

The multinode architecture is implemented in the .MPP driver. To pass parameters required for a multinode routine, you use the multinode parameter block of type `MNParamBlock`. You must define this parameter block type in your application. (See "The Multinode Parameter Block" on page 12-19.) An arrow preceding a parameter indicates whether the parameter is an input or an output parameter:

Arrow	Meaning
→	Input
←	Output

The `AddNode`, `RemoveNode`, and `NetWrite` routines use different fields of the multinode parameter block for parameters specific to the routine. The description of each routine identifies the parameter block values that the routine requires.

Assembly-language note
You call the multinode commands from assembly language by putting a routine selector in the `csCode` field of the parameter block and calling the `_Control` trap. To execute the `_Control` trap asynchronously, include the value `,ASYNC` in the operand field. Note, however, that you must execute the `AddNode` routine as an immediate (`immed`) synchronous routine. ◆

Because the MPW interface files do not define an interface for the multinode architecture, you must use the Device Manager's interface to call the multinode routines from a high-level language.

To acquire a multinode address, you execute the `AddNode` routine specifying a routine selector of 262 in the `csCode` field. You must issue the `AddNode` routine as an immediate control call to the Device Manager. See Listing 12-1 on page 12-9 for an example of how to make an immediate control call from the Pascal language.

To issue the `RemoveNode` (`csCode` equals 263) and `NetWrite` (`csCode` equals 261) routines, you use the Device Manager's `PBControl` function. The `PBControl` function is defined as follows:

```
FUNCTION PBControl (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

`paramBlock` A pointer to the multinode parameter block of type `MNParamBlock` that contains the parameters required by the multinode routine to be executed.

`async` A Boolean value that specifies whether the function is to be executed synchronously or asynchronously. Set the `async` parameter to `TRUE` to execute the function asynchronously.

DESCRIPTION

You can execute the `PBControl` function synchronously or asynchronously by setting the `async` flag. The `PBControl` function takes a pointer to a multinode parameter block that contains a `csCode` field in which you specify the routine selector for the particular routine to be executed; you must specify a numeric value for this field. You must also specify the .MPP driver reference number as the value of the multinode parameter block's `ioRefNum` field. The Device Manager's `OpenDriver` function returns the .MPP driver reference number when you call it to open the .MPP driver.

Adding and Removing Multinode Addresses

This section describes the multinode routines that you call to add or remove a multinode address for your application or process to use. You use the `AddNode` routine to add a multinode ID after you open the .MPP driver. You use the `RemoveNode` routine to remove the multinode ID when you no longer require the additional node address.

AddNode

You use the `AddNode` routine to acquire a multinode ID that is separate from and in addition to the standard user node ID assigned to the system. You call the `AddNode` routine once for each additional multinode that you require. You use the `PBControl` function to call the `AddNode` routine. See "Routines" on page 12-20 for a description of the `PBControl` function. You use a synchronous immediate control call to issue the `AddNode` routine.

Parameter block

←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The .MPP driver reference number. You must fill in this value.
→	`csCode`	`Integer`	The routine selector. Always equal to 262 for this routine. You must fill in this value.
→	`reqNodeAddr`	`AddrBlock`	The requested multinode address.
←	`actNodeAddr`	`AddrBlock`	The actual multinode address assigned and returned by the .MPP driver.
→	`recvRoutine`	`LongInt`	The address of the application's receive routine.
→	`reqCableLo`	`Integer`	The start of requested network number range for the multinode.
→	`reqCableHi`	`Integer`	The end of the requested network number range for the multinode.
→	`reserved`	`char`	70 reserved bytes required by the .MPP driver.

Field descriptions

`reqNodeAddr` The desired network address of the multinode to be acquired. You specify a value for this field in `AddrBlock` format. (See "The Address Block Record" on page 12-18.) The value of the `aSocket` field of the `AddrBlock` record must always be 0. Set the `aNet` and `aNode` fields to the desired network number and multinode ID. If the address that you specify is in use or is invalid, the .MPP driver will assign a different multinode address. To allow the .MPP driver to randomly generate the multinode address to be assigned, specify 0 for all three fields of the `AddrBlock` record. The .MPP driver returns in the `actNodeAddr` field of the parameter block either the multinode address that you request or the one that it selects.

`actNodeAddr` The actual network address of the multinode that the .MPP driver assigned and returned to you.

`recvRoutine` The address of the routine that you provide as part of your application to receive packets addressed to this multinode. The .MPP driver calls this routine when it receives either a packet addressed to the multinode or a broadcast packet.

`reqCableLo` The network number that defines the low end of the range of network numbers from which you would like the .MPP driver to select a multinode ID for your use. The `reqCableHi` field contains the network number that defines the high end of this range. The

.MPP driver uses the values that you specify for the cable range if all of the following conditions are true: the .MPP driver could not assign the multinode number that you specified in the `reqNodeAddr` field (if you specified one), there is no router on the network, and all the multinode addresses belonging to the network whose number is specified in the `NetHint` field are being used. The `NetHint` field contains the last used network number stored in RAM.

The network range for the system on which your application is running is defined by the seed router on a network.

If your application does not require that the multinode ID that the .MPP driver assigns to it belong to a specific network cable range, you can set the `reqCableLo` and `reqCableHi` fields to 0.

`reqCableHi` The network number that defines the high end of the range of network numbers from which you would like the .MPP driver to select a multinode ID for your use. The `reqCableLo` field value delimits the low end of the range.

`reserved` 70 bytes that are reserved for internal use by the .MPP driver.

DESCRIPTION

The `AddNode` routine acquires the multinode address that you specify as the value of the `reqNodeAddr` parameter if that multinode ID is available and the .MPP driver is able to service the call.

If the requested node is already in use or is invalid, or if you do not request a specific multinode ID, the .MPP driver will randomly select a multinode ID and return it as the value of the `actNodeAddr` parameter.

If the .MPP driver is unable to service the call, it will return a result code of –1021, which indicates that you should try the `AddNode` routine again. If you receive this result code, you can retry the `AddNode` routine call repeatedly until either the .MPP driver assigns and returns a multinode ID to you or you receive a different error message. Because of this need to be able to retry this call repeatedly, you cannot issue the `AddNode` call asynchronously.

Your application must provide the address of a receive routine that it uses to receive both packets addressed to the multinode and broadcast packets. You pass the address of this routine to the .MPP driver in the `recvRoutine` parameter. For more information about the receive routine, see "Receiving Packets Addressed to Your Multinode" beginning on page 12-10.

SPECIAL CONSIDERATIONS

You must issue the `AddNode` routine as a synchronous immediate control call at system task time.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `AddNode` routine from assembly language, call the `_Control` trap macro with a value of 262 in the `csCode` field of the parameter block. You must issue the routine request as an immediate call.

RESULT CODES

`noErr`	0	No error
`tryAddNodeAgainErr`	-1021	The .MPP driver was not able to add the multinode; try again
`mnNotSupported`	–1022	Multinode is not supported by the current AppleTalk connection file of type `'adev'`
`noMoreMultiNodes`	–1023	No multinode addresses are available on the network

SEE ALSO

For an example of how to issue the `AddNode` routine as a synchronous immediate control call from the Pascal language, see Listing 12-1 on page 12-9.

RemoveNode

You use the `RemoveNode` routine to remove a multinode address that you acquired through the `AddNode` routine. You use the `PBControl` function to call the `RemoveNode` routine. See "Routines" on page 12-20 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The .MPP driver reference number. You must fill in this value.
→	`csCode`	`Integer`	A routine selector. Always equal to 263 for this routine. You must fill in this value.
→	`nodeAddr`	`AddrBlock`	An address of the multinode to be removed.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, AppleTalk calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine.
If you execute a function synchronously, AppleTalk ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`nodeAddr` The address of the multinode to be removed. You specify a value for this field in `AddrBlock` format. (See "The Address Block Record" on page 12-18.) The value of the `aSocket` field of the `AddrBlock` record must always be 0. Set the `aNet` and `aNode` fields to the network number and multinode ID values of the multinode to be deleted.

DESCRIPTION

The `RemoveNode` routine removes the multinode address that you specify. You should remove only a multinode address using this routine; you must not attempt to remove the user node address.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `RemoveNode` routine from assembly language, call the `_Control` trap macro with a value of 263 in the `csCode` field of the parameter block.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter value

Sending Datagrams Through Multinodes

This section describes the `NetWrite` routine that you use to send a packet from a multinode. You can use a multinode to send a packet down through the AppleTalk protocol stack and across the AppleTalk network to another multinode or to a socket client application or process, or you can send the packet from the multinode to a socket-client application of the user node on the same system.

NetWrite

You use the `NetWrite` routine to send a packet from a multinode to another multinode or socket-client application. You use the `PBControl` function to call the `NetWrite` routine. See "Routines" on page 12-20 for a description of the `PBControl` call.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code.
→	`ioRefNum`	`Integer`	The .MPP driver reference number. You must fill in this value.
→	`csCode`	`Integer`	A routine selector. Always equal to 261 for this routine. You must fill in this value.
→	`checkSumFlag`	`Byte`	A flag indicating whether the checksum should be calculated or the existing checksum left unmodified.
→	`wdsPointer`	`Ptr`	A pointer to the write-data structure for the function.

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, AppleTalk calls your completion routine when it completes execution of the function if you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, AppleTalk ignores the `ioCompletion` field. For information about completion routines, see the chapter "Introduction to AppleTalk" in this book.

`checkSumFlag` A flag whose value you set to a nonzero number if you want the checksum for the datagram to be calculated and placed in the DDP header of the packet. If you do not want the current value in the packet header's checksum field to be altered, you set this field to 0.

`wdsPointer` A pointer to the write-data structure that contains a series of length words and pointers that indicate the length and location of a portion of the data, including the header information, that constitutes the packet to be sent over the network.

DESCRIPTION

To send a packet over an AppleTalk network from a multinode, you must first prepare a write-data structure, and then call the `NetWrite` routine, passing it a pointer to the write-data structure.

The write-data structure that you create for multinodes differs slightly from the standard write-data structure that you create to send a DDP packet using the `PWriteDDP` function. For a multinode, you must specify both the source multinode address and the destination address in the packet header information data areas that you point to from the write-data structure. You can also set the checksum field of the write-data structure to 0 to direct AppleTalk to not calculate a checksum for this packet.

You specify the source network number and the source multinode ID of the multinode; the .MPP driver does not set these values for you in the header area of a packet sent from a multinode as it does for a standard DDP packet, although both packets are transmitted as DDP datagrams.

If you are sending the contents of an existing DDP packet through the `NetWrite` call, you can leave the value of the source socket field unchanged. The value in the source socket field should adhere to the conventions that the AppleTalk DDP protocol specification describes for the use of sockets. The socket number value must fall within the defined user range as stated in the DDP protocol specification. (See *Inside AppleTalk*, second edition, for this information.)

The `checkSumFlag` parameter block field of the `NetWrite` routine relates to the standard DDP header checksum field. However, the multinode architecture uses this flag differently than the DDP interface uses it.

- If you want the checksum for the datagram to be calculated and placed in the DDP header before the .MPP driver transmits the packet, you set this field to a nonzero number.

- If you want the checksum field of the DDP packet header not to be modified, you set this field to 0, and the existing checksum value in the DDP header will not be changed.

Note that if you want to send a packet that does not include a checksum, you must hardcode the value by setting to 0 the checksum field of the data structure that contains the packet header that you point to from the write-data structure.

All packets that you send using the `NetWrite` routine are built with the long DDP packet header to allow for inclusion of the source multinode address. The DDP packet header includes the source multinode address even when the destination and source nodes are on the same LocalTalk network.

Because the source multinode ID is associated with the application that sent the packet and the source user node ID is associated with the machine that transmitted the packet, the source user node ID in the frame header and the source multinode ID in the DDP packet header are always different values.

IMPORTANT

Do not set the socket number to 0 ($00) for the source socket number that you specify in the data area pointed to by the write-data structure. You do this in the address block record socket field for the `AddNode` routine because the socket number does not apply when you are acquiring a multinode, but you must not do it for the `NetWrite` call because `NetWrite` causes the .MPP driver to build a DDP packet, and socket number 0 has special meaning to DDP that is outside the valid user socket range. ▲

SPECIAL CONSIDERATIONS

Memory used for the write-data structure belongs to the multinode implementation in the .MPP driver for the life of the `NetWrite` call and must be nonrelocatable. After the `NetWrite` call completes execution, you must release the memory that you used for the write-data structure.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `NetWrite` routine from assembly language, call the `_Control` trap macro with a value of 261 in the `csCode` field of the parameter block.

RESULT CODES

`noErr`	0	No error
`ddpLenErr`	–92	Datagram is too long
`noBridgeErr`	–93	No router found
`excessCollsns`	–95	Excessive collisions on write

SEE ALSO

See the section “Preparing a Write-Data Structure” on page 12-14 for information on how to create the write-data structure.

Summary of Multinode Architecture

The multinode architecture MPP parameter block data structure and symbolic constants for routines and result codes are not defined in the MPW interface files. (The write-data structure and the address block record are defined in the MPW interface files for use with other protocols, but you can use them for multinode also.)

You must declare the MPP parameter block for multinode in your application. If you want to use the symbolic constants for the routines and result codes, you need to declare them also.

You use the Device Manager's `PBControl` function to call the `RemoveNode` and `NetWrite` routines from the Pascal and C languages. You must issue the `AddNode` routine as an immediate synchronous control call from the Pascal and C languages. You must define a function as part of your application. (See Listing 12-1 on page 12-9 for an example of how to do this in Pascal.) From assembly language, you can directly make an `immed _Control` trap macro call.

Pascal Summary

Constants

(Declare the following constants in your application.)

```
CONST
   {csCodes}
   netWrite    = 261;              {send packet through multinode}
   addNode     = 262;              {request a multinode}
   removeNode  = 263;              {remove multinode}
```

Data Types

The Write-Data Structure

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

The Address Block Record

```
TYPE AddrBlock =
   PACKED RECORD
      aNet:          Integer;    {network number for multinode}
      aNode:         Byte;       {multinode ID}
      aSocket:       Byte;       {socket number; always 0}
   END;
```

The Multinode Parameter Block

(Declare this data type in your application.)

```
TYPEMNParmType = (AddNodeParm,RemoveNodeParm);
TYPE MNParamBlock =
   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;          {call command code}
         filler1:       Byte;          {reserved}
         checkSumFlag:  Byte;          {perform checksum on datagram}
         wdsPointer:    Ptr;           {pointer to write-data structure}
         filler2:       Integer;       {reserved}
         CASE MNParmType OF
            AddNodeParm:
            (reqNodeAddr:  AddrBlock;     {preferred address requested}
             actNodeAddr:  AddrBlock;     {actual node address returned}
             recvRoutine:  ProcPtr;       {pointer to packet receive routine}
             reqCableLo:   Integer;       {preferred network range for the }
             reqCableHi:   Integer;       { node being acquired}
             reserved:     PACKED ARRAY[1..70] OF Byte);
            RemoveNodeParm:
            (nodeAddr:     AddrBlock);    {node address to be deleted}
END;

MNParmBlkPtr = ^MNParamBlock;
```

C Summary

Constants

(Declare the following constants in your application.)

```
/*csCodes*/
enum {
   netWrite       = 261,              /*send packet through multinode*/
   addNode        = 262,              /*request a multinode*/
   removeNode     = 263               /*remove multinode*/
};
```

Data Types

The Write-Data Structure

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

The Address Block Record

```
struct AddrBlock {
      short             aNet;          /*network number for multinode*/
      unsigned char     aNode;         /*multinode ID*/
      unsigned char     aSocket;       /*socket number; always 0*/
};

typedef struct AddrBlock AddrBlock;
```

The MPP Parameter Block for Multinode

(Declare this data type in your application.)

```
typedef struct {
   MPPATPHeader
      char              filler1;       /*reserved*/
      unsigned char     checkSumFlag;  /*perform checksum on datagram*/
      Ptr               wdsPointer;    /*pointer to write-data structure*/
      char              filler2[2];    /*reserved*/
      union {
```

```
        AddrBlock      reqNodeAddr;   /*preferred address requested*/
        AddrBlock      nodeAddr;      /*node address to be deleted*/
              } MNaddrs;
      AddrBlock      actNodeAddr;      /*actual node address acquired*/
      Ptr            recvRoutine;      /*address of packet receive routine*/
      short          reqCableLo;       /*preferred network range for the */
      short          reqCableHi;       /* node being acquired*/
      char           reserved[70];
} MNParamBlock;

typedef MNParamBlock*MNParmBlkPtr;
```

Assembly-Language Summary

MPP Parameter Block Common Fields for Multinode Routines

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

AddNode Parameter Variant

26	csCode	word	routine selector; always 262 for this routine
36	reqNodeAddr	long	requested multinode address
40	actNodeAddr	long	actual multinode address assigned
44	recvRoutine	long	address of the application's receive routine
48	reqCableLo	word	beginning of requested network number range for the multinode
50	reqCableHi	word	end of the requested network number range for the multinode
52	reserved	array	70 reserved bytes required by the .MPP driver

(Note that to execute the `AddNode` routine from assembly language, you call the `_Control` trap macro and issue the routine request as an immediate call.)

RemoveNode Parameter Variant

26	csCode	word	routine selector; always 263 for this routine
36	nodeAddr	long	actual multinode address assigned

NetWrite Parameter Variant

26	`csCode`	word	routine selector; always 261 for this routine
29	`checkSumFlag`	byte	a flag indicating whether the checksum should be calculated or the existing checksum left unmodified
30	`wdsPointer`	long	a pointer to the write-data structure for this routine

Result Codes

`noErr`	0	No error
`paramErr`	–50	Bad parameter value
`ddpLenErr`	–92	Datagram is too long
`noBridgeErr`	–93	No router found
`excessCollsns`	–95	Excessive collisions on write
`tryAddNodeAgainErr`	–1021	The .MPP driver was not able to add node; try again
`mnNotSupported`	–1022	Multinode is not supported by the current AppleTalk connection file of type `'adev'`
`noMoreMultiNodes`	–1023	No node address is available on the network