Opening and Maintaining an ADSP Connection

To use the **AppleTalk DSP (ADSP)** to establish and maintain a connection between a socket on your local node and a remote socket, use the following procedure:

1. Use the **MPPOpen** function to open **The .MPP Driver**, and then use the **OpenDriver** function to open **The .DSP Driver**. The **OpenDriver** function is used to return the reference number for **The .DSP Driver**. You must supply this reference number each time you call **The .DSP Driver**.

2. Allocate nonrelocatable memory for a connection control block (**CCB**), send and receive queues, and an attention-message buffer. If you need to allocate the memory dynamically while the program is running, use the NewPtr routine. Otherwise, the way in which you allocate the memory depends on the compiler you are using. (The code example at the end of this section shows how it's done in C.) The memory that you allocate becomes the property of **ADSP** when you call the dspInit routine to establish a connection end. You cannot write any data to this memory except by calling **ADSP**, and you must ensure that the memory remains locked until you call the dspRemove routine to eliminate the connection end.

The **CCB** is 242 bytes. The attention-message buffer must be 570 bytes. When you send bytes to a remote connection end, **ADSP** stores the bytes in a buffer called the *send queue.* Until the remote connection end acknowledges their receipt, **ADSP** keeps the bytes you sent in the send queue so that they are available to be retransmitted if necessary. When the local connection end receives bytes, it stores them in a buffer, called the *receive queue,* until you read them. The sizes you need for the send and receive queues depend on the lengths of the messages being sent.

ADSP does not transmit data from the remote connection end until there is room for it in your receive queue. If your send or receive queues are too small, they limit the speed with which you can transmit and receive data. A queue size of 600 bytes should work well for most applications. If you are using **ADSP** to send a continuous flow of data, a larger data buffer improves performance. If your application is sending or receiving the user's keystrokes, a smaller buffer should be adequate. The constant minDSPQueueSize indicates the minimum queue size that you can use.

If you are using a version of **The .DSP Driver** prior to version 1.5, you must allocate send and receive queues that are 12 percent larger than the actual buffer sizes you need. You must do this in order to provide some extra space for use by **The .DSP Driver**. Version 1.5 and later versions of **The .DSP Driver** use a much smaller, and variable, portion of the buffer space for their overhead.
The .DSP Driver version number is stored in the low byte of the qFlags field, which is the first field in the dCtlQHdr field in the driver's device control entry (DCE) data structure. Version 1.5 of **The .DSP Driver** has a version number of 4 in the DCE. The DCE is described in **Device Manager** .

3.Use the dspInit routine to establish a connection end. You must provide pointers to the **CCB**, send queue, receive queue, and attention-message buffer. You may also provide a pointer to the user routine that **ADSP** calls when your connection end receives an unsolicited connection event. User routines are discussed in the section called **Writing a Connection Routine**.

If there is a specific socket that you want to use for the connection end, you can specify the socket number in the *localSocket* parameter. If you want **ADSP** to assign the socket for you, specify 0 for the *localSocket* parameter. **ADSP** returns the socket number when the dspInit routine completes execution.

4. If you wish, you can use the **NBPRegister** function to add the name and address of your connection end to the node's names table (see **NBPRegister**).

5. You can use the dspOptions routine to set several parameters that control the behavior of the connection end. Because every parameter has a default value, the use of the dspOptions routine is optional. You can specify values for the following parameters:

• The sendBlocking parameter, which sets the maximum number of bytes that may accumulate in the send queue before **ADSP** sends a packet to the remote connection end. You can experiment with different values of the sendBlocking parameter to determine which provides the best performance. Under most circumstances, the default value of 16 bytes gives good performance.

• The badSeqMax parameter, which sets the maximum number of out-of-sequence data packets that the local connection end can receive before requesting the remote connection end to retransmit the missing data. Under most circumstances, the default value of 3 provides good performance.

• The useCheckSum parameter, which determines whether the Datagram Delivery Protocol (DDP) should compute a checksum and include it in each packet that it sends to the remote connection end. Using checksums slows communications slightly. Normally **ADSP** and DDP perform enough error checking to ensure safe delivery of all data. Set the useCheckSum parameter to 1 only if you feel that the network is highly unreliable.

6. Call the dspOpen routine to open the connection. The dspOpen routine has four possible modes of operation: ocAccept, ocEstablish, ocRequest, and ocPassive. Normally you use either the ocRequest or ocPassive mode. You must specify one of these four modes for the ocMode parameter when you call the dspOpen routine.

The ocAccept mode is used only by connection servers. The ocEstablish mode is used by routines that determine their connection-opening parameters and establish a connection independently of **ADSP**, but use **ADSP** to transmit and receive data.

Use the ocRequest mode when you want to establish communications with a specific socket on the internet. When you execute the dspOpen

routine in the ocRequest mode, **ADSP** sends an open-connection request to the address you specify.

If the socket to which you send the open-connection request is a connection listener, the connection server that operates that connection listener can choose any socket on the internet to be the connection end that responds to the open-connection request. To restrict the socket from which you will accept a response to your open-connection request, specify a value for the filterAddress parameter to the dspOpen routine. When your connection end receives a response from a socket that meets the restrictions of the filterAddress parameter, it acknowledges the response and **ADSP** completes the connection.

To use the ocRequest mode, you must know the complete internet address of the remote socket, and the **ADSP** client at that address must either be a connection listener or have executed the dspOpen routine in the ocPassive mode. You can use the **PLookupName** function to obtain a list of names of objects on the internet and to determine the internet address of a socket when you know its name. The **PLookupName** function is the preferred interface version of the **NBPLookup** function. Enhancements to the wild-card-lookup feature of the Name-Binding Protocol (NBP) are discussed in **A New NBP Wildcard Character** in **PGetAppleTalkInfo**.

Use the ocPassive mode when you expect to receive an open-connection request from a remote socket. You can specify a value for the filterAddress parameter to restrict the network number, node ID, or socket number from which you will accept an open-connection request. When your connection end receives an open-connection request that meets the restrictions of the filterAddress parameter, it acknowledges the request and **ADSP** completes the connection.

You can poll the state field in the **CCB** to determine when the connection end is waiting to receive an open-connection request, when the connection end is waiting to receive an acknowledgment of an open-connection request, and when the connection is open. (see, **Connection Control Block**.) Alternatively, you can check the result code for the dspOpen routine when the routine completes execution. If the routine returns the noErr result code, then the connection is open.

7. Use the dspRead routine to read data that your connection end has received from the remote connection end. Use the dspWrite routine to send data to the remote connection end. Use the dspAttention routine to send attention messages to the remote connection end.

The dspWrite routine places data in the send queue. **ADSP** is a full-duplex, symmetric communications protocol. You can send data at any time, and your connection end can receive data at any time, even at the same time as you are sending data. **ADSP** transmits the data in the send queue when one of the following conditions occurs:

- You call the dspWrite routine with the flush parameter set to a nonzero number.

- The number of bytes in the send queue equals or exceeds the blocking

factor that you set with the dspOptions routine.

• The send timer expires. The send timer sets the maximum amount of time that can pass before **ADSP** sends all unsent data in the send queue to the remote connection end. **ADSP** calculates the best value to use for this timer and sets it automatically.

• A connection event requires that the local connection end send an acknowledgment packet to the remote connection end.

If you send more data to the send queue than it can hold, the dspWrite routine does not complete execution until it has written all the data to the send queue. If you execute the dspWrite routine asynchronously, **ADSP** returns control to your program and writes the data to the send queue as quickly as it can. This technique provides the most efficient use of the send queue by your program and by **ADSP**. Because **ADSP** does not remove data from the send queue until that data has not only been sent but also acknowledged by the remote connection end, using the flush parameter to the dspWrite routine does not guarantee that the send queue is empty. You can use the dspStatus routine to determine how much free buffer space is available in the send queue.

The dspRead routine reads data from the receive queue into your application's private data buffer. **ADSP** does not transmit data until there is space available in the other end's receive queue to accept it. Because a full receive queue slows the communications rate, you should read data from the receive queue as often as necessary to keep sufficient buffer space available for new data. You can use either of two techniques to do this:

• allocate a small receive queue (about 600 bytes) and call the dspRead routine asynchronously. Your completion routine for the dspRead routine should then call the dspRead routine again.

• allocate a large receive queue and call the dspRead routine less frequently.

If there is less data in the receive queue than the amount you specify with the *reqCount* parameter to the dspRead command, the command does not complete execution until there is enough data available to satisfy the request. There are three exceptions to this rule:

• If the end-of-message bit in the **ADSP** packet header is set, the dspRead command reads the data in the receive queue, returns the actual amount of data read in the actCount parameter, and returns the eom parameter set to 1.

• If you have closed the connection end before calling the dspRead routine (that is, the connection is half open), the command reads whatever data is available and returns the actual amount of data read in the actCount parameter.

• If **ADSP** has closed the connection before you call the dspRead routine and there is no data in the receive queue, the routine returns the noErr result code with the actCount parameter set to 0 and the eom parameter set to 0.

• In addition to the byte-stream data format implemented by the dspRead and dspWrite routines, **ADSP** provides a mechanism for sending and receiving control signals or information separate from the byte stream. You use the dspAttention routine to send an attention code and an attention message to the remote connection end. When your connection end receives an attention message, **ADSP**'s interrupt handler sets the eAttention flag in the userFlags field of the **CCB** and calls your user routine. Your user routine must first clear the userFlags field. Then your routine can read the attention code and attention message and take whatever action you deem appropriate.

• Because **ADSP** is often used by terminal emulation programs and other applications that pass the data they receive on to the user without processing it, attention messages provide a mechanism for the applications that are clients of the connection ends to communicate with each other. For example, you could use attention messages to implement a handshaking and data-checking protocol for a program that transfers disk files between two applications, neither one of which is a file server. Or a database server on a mainframe computer that uses **ADSP** to communicate with Macintosh computer workstations could use the attention mechanism to inform the workstations when the database is about to be closed down for maintenance.

• When you are ready to close the **ADSP** connection, you can use the dspClose or dspRemove routine to close the connection end. Use the dspClose routine if you intend to use that connection end to open another connection and do not want to release the memory you allocated for the connection end. Use the dspRemove routine if you are completely finished with the connection end and want to release the memory.

You can continue to read data from the receive queue after you have called the dspClose routine, but not after you have called the dspRemove routine. You can use the dspStatus routine to determine whether any data is remaining in the receive queue, or you can read data from the receive queue until both the actCount and eom fields of the dspRead parameter block return 0.

If you set the abort parameter for the dspClose or dspRemove routine to 0, then **ADSP** does not close the connection or the connection end until it has sent-and received acknowledgment for-all data in the send queue and any pending attention messages. If you set the abort parameter to 1, then **ADSP** discards any data in the send queue and any attention messages that have not already been sent.

After you have executed the dspRemove routine, you can release the memory you allocated for the **CCB** and data buffers.

The listing below (**Using ADSP to establish and use a connection**) illustrates the use of **ADSP**. This routine opens the .MPP and .DSP drivers and allocates memory for its internal data buffers, for the **CCB**, and for the send, receive, and attention-message buffers. Then the routine uses the dspInit routine to establish a connection end and uses NBP to register the name of the connection end on the internet. (The user routine specified by the userRoutine parameter to the

dspInit function is shown in the listing, in **Writing a Connection Routine**.) The code example below uses the dspOptions routine to set the blocking factor to 24 bytes. The routine uses NBP to determine the address of a socket whose name was chosen by the user and sends an open-connection request (dspOpen) to that socket. When the dspOpen routine completes execution, the routine sends data and an attention message to the remote connection end and reads data from its receive queue. Finally, the routine closes the connection end with the dspRemove routine and releases the memory it allocated.

Using ADSP to establish and use a connection (also see the section on AppleTalk DSP (ADSP).

```
// Using ADSP to establish and use a connection
// Assuming inclusion of <MacHeaders>

#include <AppleTalk.h>
#include <ADSP.h>

#define    qSize        600    // queue space
#define    myDataSize   128    // size of internal read/write buffers
#define    blockFact    24 // blocking factor

// modify the connection control block to add storage for A5

typedef struct myTRCCB {
    long    myA5;
    TRCCB u;
} myTRCCB;

void MyADSP (void);
pascal void myConnectionEvtUserRoutine (void);
void DoError (OSErr myErr);
void PickASocket (AddrBlock myAddrBlk);

Boolean     gReceivedAnEvent;
myTRCCB  gDspCCB;

void MyADSP ()
{
    unsigned char  *dspSendQPtr;
    unsigned char  *dspRecvQPtr;
    unsigned char  *dspAttnBufPtr;
    unsigned char  *myData2ReadPtr;
    unsigned char  *myData2WritePtr;
    unsigned char  *myAttnMsgPtr;

    DSPPBPtr myDSPPBPtr;
    MPPPBPtr     myMPPPBPtr;
    NamesTableEntry  myNTEName;
    AddrBlock  myAddrBlk;
    short        drvrRefNum;
    short        mppRefNum;
    short        connRefNum;
```

```
short       myAttnCode;
Byte        tempFlag;
short       tempCFlag;
OSErr       myErr;

myErr = OpenDriver("\p.MPP", &mppRefNum); // open .MPP driver
if (myErr)                          // check and handle error
    DoError(myErr);
myErr = OpenDriver("\p.DSP", &drvrRefNum); // open .DSP driver
if (myErr)                          // check and handle error
    DoError(myErr);

// allocate memory for data buffers

dspSendQPtr = (unsigned char *) NewPtr(qSize); // ADSP use only
dspRecvQPtr = (unsigned char *) NewPtr(qSize);// ADSP use only
dspAttnBufPtr = (unsigned char *) NewPtr(attnBufSize);// ADSP use only
myData2ReadPtr = (unsigned char *) NewPtr(myDataSize);
myData2WritePtr = (unsigned char *) NewPtr(myDataSize);
myAttnMsgPtr = (unsigned char *) NewPtr(myDataSize);
myDSPPBPtr = (DSPPBPtr) NewPtr(sizeof(DSPParamBlock));
myMPPPBPtr = (MPPPBPtr) NewPtr(sizeof(MPPParamBlock));

// set up dspInit parameters

myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspInit;
myDSPPBPtr->u.initParams.ccbPtr = (TPCCB) &gDspCCB;    // ptr to CCB
myDSPPBPtr->u.initParams.userRoutine = &myConnectionEvtUserRoutine;
myDSPPBPtr->u.initParams.sendQSize = qSize;     // size of send queue
myDSPPBPtr->u.initParams.sendQueue = dspSendQPtr;// send-queue buffer
myDSPPBPtr->u.initParams.recvQSize = qSize;     // size of receive queue
myDSPPBPtr->u.initParams.recvQueue = dspRecvQPtr;    // receive-queue
                                                     //buffer
myDSPPBPtr->u.initParams.attnPtr = dspAttnBufPtr; // receive-
                                                  //attention buffer
myDSPPBPtr->u.initParams.localSocket = 0;   // let ADSP assign socket

gReceivedAnEvent = FALSE;
gDspCCB.myA5 = SetCurrentA5();     // save A5 for the user routine

// establish a connection end

myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);

if (myErr) // check and handle error
    DoError(myErr);

connRefNum = myDSPPBPtr->ccbRefNum;  // save CCB ref num for later

NBPSetNTE((Ptr) &myNTEName, (Ptr) "\pThe Object",
        (Ptr) "\pThe Type", (Ptr) "\p*",
        myDSPPBPtr->u.initParams.localSocket);
        // set up NBP names table entry

// set up PRegisterName parameters
```

```
myMPPPBPtr->NBP.interval = 7;// retransmit every 7*8=56 ticks
myMPPPBPtr->NBP.count = 3;  // and retry 3 times
myMPPPBPtr->NBP.NBPPtrs.entityPtr = (Ptr) &myNTEName;
                        // name to register
myMPPPBPtr->NBP.parm.verifyFlag = 0;  // don't verify this name

myErr = PRegisterName(myMPPPBPtr, FALSE);// register this socket

if (myErr)
    DoError(myErr);                 // check and handle error

// set up dspOptions parameters

myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspOptions;
myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
myDSPPBPtr->u.optionParams.sendBlocking = blockFact;
                        // quantum for data packet
myDSPPBPtr->u.optionParams.badSeqMax = 0;// use default
myDSPPBPtr->u.optionParams.useCheckSum = 0;  // don't calculate
                                             //checksum

myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);
                        // set options

if (myErr)
    DoError(myErr);         // check and handle error

    // routine using the PLookupName function to pick a socket
    // that will be used to establish an open connection

PickASocket(myAddrBlk);

// open a connection with the chosen socket

// set up dspOpen parameters

myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspOpen;
myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
myDSPPBPtr->u.openParams.remoteAddress = myAddrBlk;
            //address of remote socket from PLookupName function

myDSPPBPtr->u.openParams.filterAddress = myAddrBlk;
                // address filter, specified socket address only

myDSPPBPtr->u.openParams.ocMode = ocRequest;  // open connection
                        //mode
myDSPPBPtr->u.openParams.ocInterval = 0;    // use default retry interval
myDSPPBPtr->u.openParams.ocMaximum = 0;
                        // use default retry maximum

myErr = PBControl((ParmBlkPtr)myDSPPBPtr, FALSE);
                        // open a connection
if (myErr != noErr)
```

```
    DoError(myErr);          // check and handle error

// the connection with the chosen socket is open, so now send to
//  the send queue exactly myDataSize number of bytes

// set up dspWrite parameters
myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspWrite;
myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
myDSPPBPtr->u.ioParams.reqCount = myDataSize;
                        // write this numberof bytes
myDSPPBPtr->u.ioParams.dataPtr = myData2WritePtr;   // pointer to
                                    // send queue
myDSPPBPtr->u.ioParams.eom = 1;// 1 means last byte is logical
                        // end-of-message
myDSPPBPtr->u.ioParams.flush = 1;    // 1 means send data now

myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);
                        // send data to the remote connection
if (myErr)
    DoError(myErr);          // check and handle error

// now send an attention message to the remote connection end

// set up dspAttention parameters
myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspAttention;
myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
myDSPPBPtr->u.attnParams.attnCode = 0;// user-defined attention code
myDSPPBPtr->u.attnParams.attnSize = myDataSize;    // length of attention
                                    // message
myDSPPBPtr->u.attnParams.attnData = myAttnMsgPtr;
                                    // attention message

myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);
if (myErr)                  // check and handle error
    DoError(myErr);

// Now read from the receive queue exactly myDataSize number of bytes.

// set up dspRead parameters
myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
myDSPPBPtr->csCode = dspRead;
myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
myDSPPBPtr->u.ioParams.reqCount = myDataSize;
                        // read this number of bytes
myDSPPBPtr->u.ioParams.dataPtr = myData2ReadPtr;     // pointer to
                                    // read buffer

myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);
                    // read data from the remote  connection
if (myErr)                  // check and handle error
    DoError(myErr);

// we're done with the connection, so remove it
```

```
    // set up dspRemove parameters

    myDSPPBPtr->ioCRefNum = drvrRefNum;  // ADSP driver ref num
    myDSPPBPtr->csCode = dspRemove;
    myDSPPBPtr->ccbRefNum = connRefNum;  // connection ref num
    myDSPPBPtr->u.closeParams.abort = 0;     // don't close until everything
                                    //is sent and received

    myErr = PBControl((ParmBlkPtr) myDSPPBPtr, FALSE);
                                    // close and remove the connection
    if (myErr)
        DoError(myErr);         // check and handle error

    // you're done with this connection, so give back the memory
    DisposPtr((Ptr) dspSendQPtr);
    DisposPtr((Ptr) dspRecvQPtr);
    DisposPtr((Ptr) dspAttnBufPtr);
    DisposPtr((Ptr) myData2ReadPtr);
    DisposPtr((Ptr) myData2WritePtr);
    DisposPtr((Ptr) myAttnMsgPtr);
    DisposPtr((Ptr) myDSPPBPtr);
    DisposPtr((Ptr) myMPPPBPtr);
}   // MyADSP
```