This chapter describes the AppleTalk Data Stream Protocol (ADSP) that you use to establish a session to exchange data between two network processes or applications in which both parties have equal control over the communication. You should read this chapter if you want to write an application that supports the exchange of more than a small amount of data between two parties who each can both send and receive streams of data.

This chapter also describes the AppleTalk Secure Data Stream Protocol (ASDSP), a secure version of ADSP, that allows users of your application to communicate over an ADSP session after the users' identities have been authenticated. Users can then exchange encrypted data over the session. For your application to use ASDSP, the system on which it runs must have the AppleTalk Open Collaboration Environment (AOCE) software installed and must have access to an AOCE server. To use ASDSP, you must also use the Authentication Manager, which is a component of the AOCE software. For information on the Authentication Manager, refer to *Inside Macintosh: AOCE Application Programming Interfaces.*

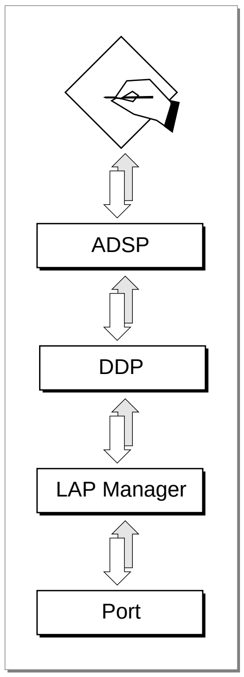
ASDSP enhances ADSP with authentication and encryption features. When this chapter discusses components of ADSP, such as connection ends and connection listeners, you can assume that the information also applies to ASDSP. The sections in this chapter that discuss ASDSP describe any specific differences between it and the standard version of ADSP. To use ASDSP, you should be familiar with ADSP.

For an overview of ADSP and how it fits within the AppleTalk protocol stack, read the chapter "Introduction to AppleTalk" in this book, which also introduces and defines some of the terminology used in this chapter. For a complete explanation of the ADSP specification, see *Inside AppleTalk,* second edition.

About ADSP

ADSP includes both session and transport services, and it is the most commonly used of the AppleTalk transport protocols. The .DSP driver implements ADSP. ADSP allows you to establish and maintain a connection between two AppleTalk network entities and transfer data across this connection as a continuous stream. Because ADSP is a client of DDP, data that you transmit using ADSP is actually sent and received over the AppleTalk internet in packets. However, ADSP builds a session connection on top of the packet transfer services that DDP provides so that applications using ADSP can exchange data as a continuous stream. Figure 5-1 on page 5-4 shows ADSP and the underlying protocols that it uses; ADSP is a client of DDP, just as your application is a client of ADSP.

Figure 5-1 ADSP and its underlying protocols



Communication between two applications using ADSP occurs over a connection that is made between the two sockets that these network entities use; ADSP assigns a socket to be used when you initialize each end of the connection, and your application becomes a client of that socket. Because this connection exists for the duration of the exchange, ADSP is called a *connection-oriented* protocol. ADSP manages and controls the data flow between the two sockets throughout the session to ensure that

- the data is delivered and received in the order in which it was sent
- duplicate data is not sent
- the application or process at the receiving end of the connection has the buffer capacity to accept the data

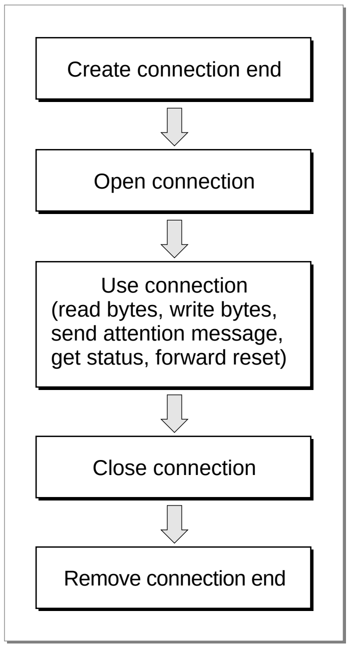
In an ADSP session, both ends of the connection have equal control over the communication in a ***peer-to-peer*** relationship. For the two ends of an ADSP connection to function properly, each must maintain information to control the connection and determine the connection state. To accommodate these requirements, the socket at either end of the connection has associated with it information that defines the state of the connection and information that the application and ADSP use to control the connection and communicate over it. The combination of a socket and the ADSP information maintained by the socket client is referred to as a ***connection end.*** To create a connection, two connection ends must be set up and initialized. Each connection end views itself as the local end and the other as the remote end.

Your application can use ADSP to

- create a connection end
- request a connection with a remote connection end
- create a ***connection listener*** to wait passively for connection requests from remote connection ends (see "Connection Listeners" on page 5-7 for more information)
- read data from and write it to an open connection
- close a connection without removing it
- remove a connection end

Figure 5-2 shows the order in which applications commonly call the ADSP routines to perform these functions for a connection end. (Figure 5-4 on page 5-8 shows this for a connection listener.)

Figure 5-2 Steps for creating an ADSP connection end


Create connection end
Open connection
Use connection (read bytes, write bytes, send attention message, get status, forward reset)
Close connection
Remove connection end

ADSP provides for a full-duplex data stream between the two ends of the connection that allows for a full-duplex dialog; this means that either end of the connection can call routines to send data at any time. (However, full-duplex does not mean that both connection ends actually send electrical signals at the same time; ADSP controls this process.) See the chapter "Introduction to AppleTalk" in this book for more information on full-duplex communication.

In addition to the full-duplex data stream that an ADSP session maintains, ADSP allows either end of a connection to send an attention message to the other end without interrupting the primary flow of data.

Among the features that ADSP provides are

- an end-of-message feature that lets you break streams of data into logical messages
- an attention-message feature that lets you and your partner application signal to each other outside the normal exchange of data
- a forward-reset feature that lets you cancel the delivery of any data that is in your connection end's send queue and any data that you have sent that is in transit and that the remote connection end has not received
- a built-in flow control feature that ensures that your application sends data only if its remote partner has the buffer capacity to receive it

Connections, Connection Ends, and Connection States

A connection is an association between two sockets that supports the flow of data between the clients of those sockets in a reliable way. Each socket can maintain concurrent ADSP connections with several other sockets, but there can be only one ADSP connection between any two sockets at one time. For example, a single socket on node A can have multiple concurrent sessions consisting of one connection to a socket on node B, one connection to a socket on node C, and one connection to a socket on node D.

When you establish an ADSP connection end, you allocate a nonrelocatable block of memory called a ***connection control block (CCB)*** in which ADSP stores state information about the connection end. When you initialize the connection end, ADSP uses the CCB to set up control information that it maintains and uses for synchronizing communication with the other socket client and for error checking.

You can read the CCB fields to gain information about the current state of the connection end. In addition to the unique AppleTalk internet address associated with a socket, each instance of a connection end has associated with it a connection ID that identifies it. You can open a connection for a socket and close that connection without actually removing the connection end, and then open another connection for the same socket. When you close a connection, the socket number remains associated with the connection, as do the data structures whose memory you allocated. ADSP uses this to ensure that any data meant for the old connection end is not delivered to the new connection end using the same socket number and data structures.

ADSP cannot deliver packets to a connection end based on the AppleTalk internet socket address alone. The connection ID ensures that a packet is delivered to the specific connection end for which it was intended. You call the new connection ID (`dspNewCID`) routine to cause ADSP to assign a connection ID to the connection end before you open a connection. ADSP assigns a connection ID number, which it includes in every packet that it delivers from your connection end to a remote connection end.

Figure 5-3 ADSP connection ends and their components

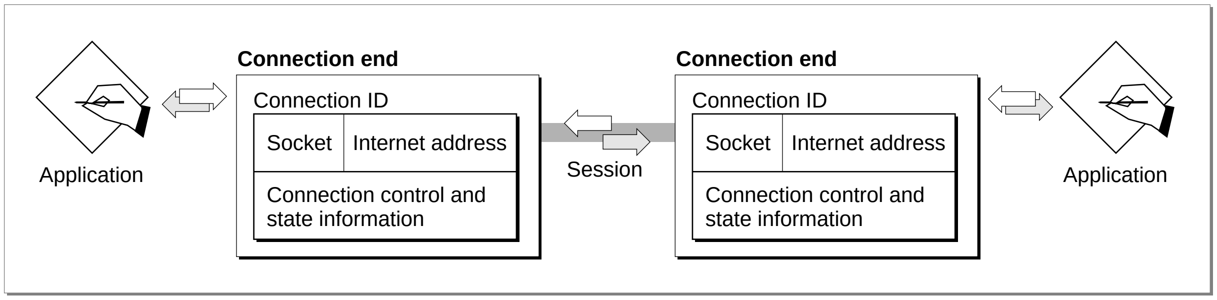


Figure 5-3 shows two connection ends and the client applications that use them to participate in a session with each other over an ADSP connection. This figure shows the components that constitute a connection end.

At any time, either end of a potential ADSP connection can initiate a session. Also, either end of the connection can tear down the connection when it is no longer needed.

- When two connection ends establish communication, the connection is considered an ***open connection.***
- When both connection ends terminate the connection and dispose of the connection information each maintains, the connection is considered a ***closed connection.***
- If one connection end is established but the other connection end is unreachable or has disposed of its connection information, the connection is considered a ***half-open connection.***

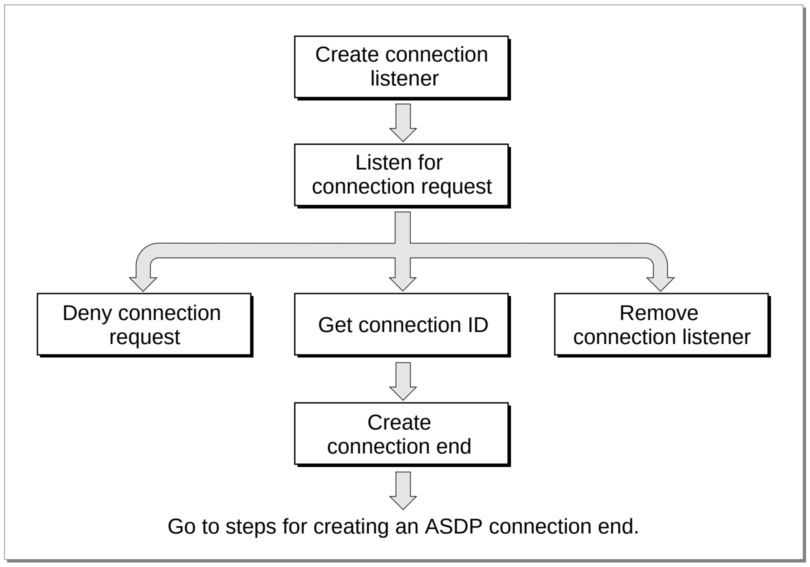
No communication can occur over a half-open or closed connection.

To prevent a half-open connection from tying up resources, ADSP automatically closes any half-open connection that cannot reestablish communication within two minutes and informs its client that the connection is closed. Under these circumstances, ADSP will call the application-supplied completion routine for any pending asynchronous ADSP routine, if one was provided. Otherwise, the pending ADSP routine will return to the calling program with an `errState` error message. If you attempt to call an ADSP routine on a half-open connection, ADSP also returns the `errState` error message.

Connection Listeners

A connection listener or a connection-listening socket is a socket that accepts open-connection requests and passes them along to its client, a connection server process, for further processing. The server then selects a socket and requests ADSP to open a connection using that socket. The connection listener can also deny an open-connection request. By specifying filtering values for the network address of the requester, you can control which requests are accepted or denied. The use of a connection listener is typical of a server environment in which a server, such as a file server, is registered with NBP using a single name. Various connection ends throughout the network contact the server's connection listener with open-connection requests. The connection server can honor the requests, or it can deny them. It might deny a request, for example, when its resources are exhausted. Figure 5-4 shows the tasks for an ADSP connection listener in the order that applications commonly perform them.

Figure 5-4 Standard tasks for an ADSP connection listener


Create connection listener
Listen for connection request
Deny connection request
Get connection ID
Remove connection listener
Create connection end
Go to steps for creating an ASDP connection end.

Reliable Delivery of Data

ADSP guarantees that data bytes are delivered in the same order as they were sent and that they are free of duplicates. It ensures that all data sent is delivered to the remote connection end's receive buffer. To accomplish this, ADSP associates a sequence number with each byte that it sends. ADSP discards any out-of-sequence data or any duplicates that are delivered. ADSP uses the sequence numbers to ensure that all of the data that one end sends is received by the other end. If data is lost, ADSP retransmits it. ADSP can send the data again because the data remains in the sending connection end's send queue until the remote end actually receives a copy of it. For more information about how ADSP delivers data, see *Inside AppleTalk,* second edition.

Unsolicited ADSP Events

After you open a connection, you can receive events that are not generated in response to any of the ADSP calls that your application makes. The other connection end or ADSP initiates these events. For example, the remote connection end can send you an attention message or a forward reset.

You receive a ***forward reset*** event when the remote connection end cancels delivery of all outstanding data to your connection end. A forward reset causes ADSP to discard all data in the send queue, all data in transit to the remote connection end, and all data in the remote connection end's receive queue that the client has not yet read.

The remote connection end can close the connection, and this, too, will generate an event notification for your connection end. You also receive event notification when ADSP tears down a connection because the remote end has become unreachable.

ADSP sets the bits of your connection end's connection control block user flags field to identify the type of event. For more information about this field, see "Creating and Using a Connection Control Block" on page 5-12. You can provide a user routine that ADSP is to call whenever you receive one of these events. This user routine is similar in concept and use to an `ioCompletion` routine that many of the other AppleTalk protocols use. See "Writing a User Routine for Connection Events" on page 5-26 for information on how to write a user routine.

About ASDSP

This section describes the secure version of ADSP referred to as ***AppleTalk Secure Data Stream Protocol (ASDSP).*** ASDSP is a superset of ADSP that includes authentication and encryption features. To use ASDSP, you should be familiar with both ADSP and ASDSP.

ASDSP features allow you to provide users of your application with the ability to exchange encrypted data across a ***secure session*** that is established after the users' identities are proven through what is known as the ***authentication process.*** Before transmitting the data that a user sends, ASDSP encrypts it and then decrypts the data before delivering it to the application at the remote connection end. Users might want to identify one another, for example, to verify that a piece of electronic mail came from the sender who claimed to be its author, and they might want to encrypt data that traverses a network if that data is considered confidential or private and they do not want others to intercept and read the data.

To verify the identities of two ends of a connection, an ASDSP application relies on information that is provided by an Apple Open Collaboration Environment (AOCE) authentication server. Your ASDSP client application at the connection end that initiates the session calls the AOCE Authentication Manager to acquire the information necessary for the authentication process from the authentication server, and then it passes this information on to ASDSP.

Note
Because ASDSP is dependent on information from the authentication server, your ASDSP application can only run on systems that also run AOCE and that have access to an AOCE authentication server. If the AOCE software is installed on the system that runs your application and if the system has access to an AOCE authentication server, your application can use ASDSP. ◆

You perform the first part of the authentication process by requesting information from the authentication server and giving that information to ASDSP to transmit to the other end of the connection. The authentication process culminates in a challenge-and-reply handshake that the ASDSP code performs on behalf of your ASDSP client application at each end of the connection to ensure that the application users are who they claim to be. The ASDSP client application of the connection end that retrieves the information from the authentication server and makes the request to open the session is called the ***initiator;*** the ASDSP client application of the connection end that receives the request and the information from the server is called the ***recipient.***

The Authentication Process

This section describes the general strategy of the authentication process. Understanding what this process entails can be helpful in understanding the meaning and use of the parameters that you get from the authentication server and pass to ASDSP.

The initiator and the recipient each have a ***private key.*** The private key, also called a *user key* or *client key,* is a number that is derived from a password; the number is used by an encryption algorithm.

The initiator calls the authentication server to request information and ***credentials*** to be used by ASDSP in establishing an authenticated session. The credentials contain information that is required in order to prove that the users of both ends of the connection are who they claim to be. The user of the initiator ASDSP client application gives the authentication server his own name or identity and that of the user of the recipient ASDSP client application.

The authentication server returns to the initiator a unique session key that the server generates exclusively for use by the authentication process for this session; the session key is valid for a limited time only. The authentication server also returns to the initiator a set of credentials that are encrypted in the recipient's private key. The credentials contain the session key also and the initiator's identity, as well as the identity of an ***intermediary*** or proxy, if one was used to obtain the credentials from the authentication server.

The initiator passes a block of data containing the credentials to ASDSP, and ASDSP on the initiator's end sends the credentials to ASDSP on the recipient's end. The latter decrypts the entire credentials block, obtaining the session key from the credentials block. ASDSP on the recipient's end then uses the session key in the authentication process that it performs on behalf of the recipient. ASDSP has the recipient's private key, which it uses to decrypt the credentials. If the authentication process succeeds, ASDSP returns all of the credentials to the recipient.

Because the initiator and ASDSP on behalf of the recipient must each decrypt the session key using their own private key, they can each be convinced that the other is who they claim to be if they can conclude that the other knows the session key. The need for this conviction begins the challenge-and-reply authentication process that enables each end to confirm that the other end also knows the unique session key.

ASDSP performs the challenge-and-reply process on behalf of the client applications in a manner that is transparent to the applications. If the authentication process completes successfully, ASDSP opens a secure connection; if the authentication process fails, ASDSP returns an error code to both the initiator and the recipient and tears down the connection that was established to perform the authentication process. To learn more about the challenge-and-reply process, see the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Programming Interfaces*.

The Data Encryption Feature

After ASDSP successfully completes the authentication process, the two ends of the connection whose identities have been verified can exchange data and they can also encrypt that data. The ASDSP encryption feature allows each party to send data that can be trusted to be securely transmitted in a manner that is unreadable by anyone other than the intended recipient until that data is decrypted by ASDSP and delivered to the recipient at the other end of the ASDSP session connection. ASDSP encrypts only data in the main data stream; it does not encrypt data in attention messages or ASDSP packet headers.

Using ADSP

This section describes how to use ADSP to

- open and maintain an ADSP connection, including how to
  - initialize the connection end (`dspInit`)
  - set options that control the behavior of the connection end (`dspOptions`)
  - open the connection (`dspOpen`)
  - read (`dspRead`) and write (`dspWrite`) data over the connection
  - send an attention code and an attention message to the remote connection end (`dspAttention`)
  - close the connection (`dspClose`) and remove it (`dspRemove`)
- create and use a connection listener, including how to
  - initialize a connection listener (`dspCLInit`)
  - activate the connection listener, causing it to listen for an open-connection request (`dspCLListen`), filtering requests that you will accept by restricting network addresses
  - initialize (`dspInit`) and open (`dspOpen`) a connection end in response to an open request that you want to accept
  - read (`dspRead`) and write (`dspWrite`) data over the connection and close the connection (`dspClose`)
  - remove the connection listener when you are finished with it (`dspCLRemove`)
- handle unsolicited connection events using your own user routine

You execute ADSP routines by calling the Device Manager's `PBControl` function. When you call the `PBControl` function for an ADSP routine, you provide a pointer to a parameter block of type `DSPParamBlock`.

You use the parameter block fields to specify the input parameters that ADSP requires to execute the command. The parameter block also includes fields whose values ADSP returns. For a complete description of the DSP parameter block and its fields, see "The DSP Parameter Block" beginning on page 5-38.

Allocating Memory for ADSP

To open and maintain an ADSP session, you must allocate memory required for the session. Depending on the ADSP routine that you call, you must allocate the following:

- storage of the state information that ADSP maintains at either end of a connection (see the discussion of the connection control block in "Connections, Connection Ends, and Connection States" on page 5-6)
- a parameter block that you use to pass parameters when you execute an ADSP routine
- a send queue and a receive queue
- an attention message buffer

This memory belongs to ADSP until you explicitly remove the connection end.

Creating and Using a Connection Control Block

When you establish an ADSP connection end, you must allocate a nonrelocatable block of memory for (and provide a pointer to) a connection control block (CCB) data structure, which ADSP uses to store state information about the connection end. This memory belongs to ADSP until you explicitly remove the connection end using the `dspRemove` routine (see "dspRemove" on page 5-62). Only then can you release or reuse the memory that you allocated for the CCB.

Most of the fields of the CCB are for ADSP's internal use. Although you must not alter any of the CCB fields except one, the `userFlags` field, you may poll them to gain information about the current state of the connection end.

When your connection end receives an unsolicited event, such as an attention message or a forward reset, ADSP's interrupt handler sets a bit corresponding to the event type in the `userFlags` field and calls your user routine, if you provided one. If you did not provide a user routine, you can test these bits to determine when an unsolicited event occurs on the connection end.

After you read them, you must clear the bits either through your user routine or directly before you handle the event.

The CCB is a record of type `TRCCB` that must consist of 242 bytes. See "The ADSP Connection Control Block Record" beginning on page 5-35 for a description of the CCB and the fields that comprise it.

Opening and Maintaining an ADSP Connection

To use 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 Device Manager's `OpenDriver` function to open the .MPP driver, and then use it again to open the .DSP driver. The .MPP driver must be open before you open the .DSP driver. The `OpenDriver` function call for the .DSP driver returns the driver reference number. You must supply this reference number each time you call the Device Manager's `PBControl` function to execute an ADSP routine.
2. Allocate nonrelocatable memory for a 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. (Listing 5-1 on page 5-17 shows how to do this in Pascal.) 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`, which is defined in the MPW interface file for ADSP, 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 buffer space for 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. See the chapter "Device Manager" in *Inside Macintosh: Devices* for information on the DCE.
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 a user routine that ADSP calls when your connection end receives an unsolicited connection event. See the section"Writing a User Routine for Connection Events" on page 5-26 for information on providing a user 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; in this case, ADSP returns the socket number when the `dspInit` routine completes execution.

4. If you wish, you can use the Name-Binding Protocol (NBP) routines to add the name and address of your connection end to the node's names table. See the chapter "Name-Binding Protocol (NBP)" in this book for information on NBP.
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 AppleTalk 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 select 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 NBP routines 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. See the chapter "Name-Binding Protocol (NBP)" in this book for information on the NBP routines.

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 the section "The ADSP Connection Control Block Record" beginning on page 5-35 for a description of the CCB fields. 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 been not only 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.

8. 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.

Listing 5-1 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 Listing 5-3 on page 5-28.) Next, Listing 5-1 uses the `dspOptions` routine to set the blocking factor to 24 bytes. This routine then uses NBP to determine the address of a socket whose name was selected by the user and sends an open-connection request (`dspOpen`) to that socket. When the `dspOpen` routine completes execution, it 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.

Listing 5-1 Using ADSP to establish and use a connection

```
PROCEDURE MyADSP;

CONST
   qSize =  600;                         {queue space}
   myDataSize =   128;                   {size of internal read/write buffers}
   blockFact = 24;                       {blocking factor}

TYPE
{Modify the connection control block to add storage for A5.}
myTRCCB =
   RECORD
      myA5: LongInt;
      u: TRCCB;
   END;

VAR
   dspSendQPtr:      Ptr;
   dspRecvQPtr:      Ptr;
   dspAttnBufPtr:    Ptr;
   myData2ReadPtr:   Ptr;
   myData2WritePtr:  Ptr;
   myAttnMsgPtr:     Ptr;
   dspCCB:           myTRCCB;
   myDSPPBPtr:       DSPPBPtr;
   myMPPPBPtr:       MPPPBPtr;
   myNTEName:        NamesTableEntry;
   myAddrBlk:        AddrBlock;
   drvrRefNum:       Integer;
   mppRefNum:        Integer;
   connRefNum:       Integer;
   gReceivedAnEvent: Boolean;
   myAttnCode:       Integer;
   tempFlag:         Byte;
   tempCFlag:        Integer;
   myErr:            OSErr;

BEGIN
   myErr := OpenDriver('.MPP', mppRefNum);   {open .MPP driver}
   IF myErr <> noErr THEN DoErr(myErr);      {check and handle error}
   myErr := OpenDriver('.DSP', drvrRefNum);  {open .DSP driver}
   IF myErr <> noErr THEN DoErr(myErr);      {check and handle error}

   {Allocate memory for data buffers.}
   dspSendQPtr := NewPtr(qSize);             {ADSP use only}
   dspRecvQPtr := NewPtr(qSize);             {ADSP use only}
   dspAttnBufPtr := NewPtr(attnBufSize);     {ADSP use only}
   myData2ReadPtr := NewPtr(myDataSize);
   myData2WritePtr := NewPtr(myDataSize);
   myAttnMsgPtr := NewPtr(myDataSize);
   myDSPPBPtr := DSPPBPtr(NewPtr(SizeOf(DSPParamBlock)));
   myMPPPBPtr := MPPPBPtr(NewPtr(SizeOf(MPPParamBlock)));

   WITH myDSPPBPtr^ DO                       {set up dspInit parameters}
   BEGIN
      ioCRefNum := drvrRefNum;               {ADSP driver ref num}
      csCode := dspInit;
      ccbPtr := @dspCCB;                     {pointer to CCB}
      userRoutine := @myConnectionEvtUserRoutine;
                                             {see Listing 5-3}
      sendQSize := qSize;                    {size of send queue}
      sendQueue := dspSendQPtr;              {send-queue buffer}
      recvQSize := qSize;                    {size of receive queue}
      recvQueue := dspRecvQPtr;              {receive-queue buffer}
      attnPtr := dspAttnBufPtr;              {receive-attention buffer}
      localSocket := 0;                      {let ADSP assign socket}
   END;

   gReceivedAnEvent := FALSE;
   dspCCB.myA5 := SetCurrentA5;              {save A5 for the user routine}
   {Establish a connection end.}
   myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
   IF myErr <> noErr THEN DoErr(myErr);
                                             {check and handle error}
   connRefNum := myDSPPBPtr^.ccbRefNum;
                                             {save CCB ref num for later}

   NBPSetNTE(@myNTEName, 'The Object', 'The Type',
             '*', myDSPPBPtr^.localSocket);
                                             {set up NBP names table entry}
   WITH myMPPPBPtr^ DO                       {set up PRegisterName }
                                             { parameters}
   BEGIN
      interval := 7;                         {retransmit every 7*8=56 ticks}
      count := 3;                            {retry 3 times}
      entityPtr := @myNTEName;               {name to register}
      verifyFlag := 1;                       {verify this name}
   END;
   {Register this socket.}
   myErr := PRegisterName(myMPPPBPtr, FALSE);
                                             {register this socket}
   IF myErr <> noErr THEN DoErr(myErr);
                                             {check and handle error}

   WITH myDSPPBPtr^ DO                       {set up dspOptions parameters}
   BEGIN
      ioCRefNum := drvrRefNum;               {ADSP driver ref num}
      csCode := dspOptions;
      ccbRefNum := connRefNum;               {connection ref num}
      sendBlocking := blockFact;             {quantum for data packet}
      badSeqMax := 0;                        {use default}
      useCheckSum := 0;                      {don't calculate checksum}
   END;
   myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                             {set options}
   IF myErr <> noErr THEN DoErr(myErr);
                                             {check and handle error}
```

```
PickASocket(myAddrBlk);                   {routine using the PLookupName }
                                          { function to pick a socket }
                                          { for the connection}
{Open a connection with the selected socket.}
WITH myDSPPBPtr^ DO                       {set up dspOpen parameters}
BEGIN
   ioCRefNum := drvrRefNum;               {ADSP driver ref num}
   csCode := dspOpen;
   ccbRefNum := connRefNum;               {connection ref num}
   remoteAddress := myAddrBlk;            {address of remote socket }
                                          { from PLookupName function}
   filterAddress := myAddrBlk;            {address filter,specified }
                                          { socket address only}
   ocMode := ocRequest;                   {open connection mode}
   ocInterval := 0;                       {use default retry interval}
   ocMaximum := 0;                        {use default retry maximum}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                          {open a connection}
IF myErr <> noErr THEN DoErr(myErr);      {check and handle error}

{The connection with the selected socket is open, so now send }
{ to the send queue exactly myDataSize number of bytes.}
WITH myDSPPBPtr^ DO                       {set up dspWrite parameters}
BEGIN
   ioCRefNum := drvrRefNum;               {ADSP driver ref num}
   csCode := dspWrite;
   ccbRefNum := connRefNum;               {connection ref num}
   reqCount := myDataSize;                {write this number of bytes}

   dataPtr := myData2WritePtr;            {pointer to send queue}
   eom := 1;                              {1 means last byte is }
                                          { logical end-of-message}
   flush := 1;                            {1 means send data now}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                          {send data to the remote }
                                          { connection}
IF myErr <> noErr THEN DoErr(myErr);
                                          {check and handle error}
```

```
{Now send an attention message to the remote connection end.}
WITH myDSPPBPtr^ DO                         {set up dspAttention parameters}
BEGIN
   ioCRefNum := drvrRefNum;                 {ADSP driver ref num}
   csCode := dspAttention;
   ccbRefNum := connRefNum;                 {connection ref num}
   attnCode := 0;                           {user-defined attention code}
   attnSize := myDataSize;                  {length of attention message}
   attnData := myAttnMsgPtr;                {attention message}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
IF myErr <> noErr THEN DoErr(myErr);
                                            {check and handle error}

{Now read from the receive queue exactly myDataSize number }
{ of bytes.}
WITH myDSPPBPtr^ DO                         {set up dspRead parameters}
BEGIN
   ioCRefNum := drvrRefNum;                 {ADSP driver ref num}
   csCode := dspRead;
   ccbRefNum := connRefNum;                 {connection ref num}
   reqCount := myDataSize;                  {read this number of bytes}
   dataPtr := myData2ReadPtr;               {pointer to read buffer}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                            {read data from the remote }
                                            { connection}
IF myErr <> noErr THEN DoErr(myErr);        {check and handle error}

{We're finished with the connection, so remove it.}
WITH myDSPPBPtr^ DO                         {set up dspRemove parameters}
BEGIN
   ioCRefNum := drvrRefNum;                 {ADSP driver ref num}
   csCode := dspRemove;

   ccbRefNum := connRefNum;                 {connection ref num}
   abort := 0;                              {don't close until }
                                            { everything is sent and }
                                            { received}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                            {close and remove the }
                                            { connection}
IF myErr <> noErr THEN DOErr(myErr);
                                            {check and handle error}
   {You're finished with this connection, so release the memory.}
   DisposPtr(dspSendQPtr);
   DisposPtr(dspRecvQPtr);
   DisposPtr(dspAttnBufPtr);
   DisposPtr(myData2ReadPtr);
   DisposPtr(myData2WritePtr);
   DisposPtr(myAttnMsgPtr);
   DisposPtr(Ptr(myDSPPBPtr));
   DisposPtr(Ptr(myMPPPBPtr));

END;         {MyADSP}
```

Creating and Using a Connection Listener

A connection listener is a special sort of ADSP connection end that cannot receive or transmit data streams or attention messages. The sole function of a connection listener is to wait passively to receive an open-connection request and to inform its client, the connection server, when it receives one. The connection server can then accept or deny the open-connection request. If it accepts the request, the connection server selects a socket to use as a connection end, establishes a connection end on that socket, and sends an acknowledgment and connection request back to the requesting connection end. The connection server can use the same socket as it used for the connection listener, or it can select a different socket as the connection end.

Use the following procedure to establish a connection listener and to use that connection listener to open a connection with a remote connection end:

1. Use the Device Manager's `OpenDriver` function to open the .MPP driver and then use the `OpenDriver` function to open the .DSP driver. The `OpenDriver` function returns 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, which is described in "Connections, Connection Ends, and Connection States" on page 5-6. The CCB is 242 bytes. A connection listener does not need send and receive queues or an attention-message buffer. The memory that you allocate becomes the property of ADSP when you call the `dspCLInit` routine to establish a connection listener. 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.
3. Call the `dspCLInit` routine to establish a connection listener. You must provide a pointer to the CCB.

   If there is a specific socket that you want to use for the connection listener, 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 `dspCLInit` routine completes execution.

4. If you wish, you can use the NBP routines to add the name and address of your connection listener to the node's names table. See the chapter "Name-Binding Protocol (NBP)" in this book for information on NBP.

5. Use the `dspCLListen` routine to cause the connection listener to wait for an open-connection request. Because the `dspCLListen` routine does not complete execution until it receives a connection request, you should call this routine asynchronously. 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 the `dspCLListen` routine receives an open-connection request that meets the restrictions of the `filterAddress` parameter, it returns a `noErr` result code (if you executed the routine asynchronously, it places a `noErr` result code in the `ioResult` parameter) and places values in the parameter block for the `remoteCID`, `remoteAddress`, `sendSeq`, `sendWindow`, and `attnSendSeq` parameters.

6. If you want to open the connection, call the `dspInit` routine to establish a connection end. You can use any available socket on the node for the connection end, including the socket that you used for the connection listener. Because a single socket can have more than one CCB connected with it, the socket can function simultaneously as a connection end and a connection listener.

   You can check the address of the remote socket to determine if it meets your criteria for a connection end. Although the `filterAddress` parameter to the `dspCLListen` routine provides some screening of socket addresses, it cannot check for network number ranges, for example, or for a specific set of socket numbers. If for some reason you want to deny the connection request, call the `dspDeny` routine, specifying the CCB of the connection listener in the `ccbRefNum` parameter. Because the `dspCLListen` routine completes execution when it receives an open-connection request, you must return to step 5 to wait for another connection request.

7. Call the `dspOpen` routine to open the connection. Specify the value `ocAccept` for the `ocMode` parameter and specify in the `ccbRefNum` parameter the reference number of the CCB for the connection end that you want to use. When you call the `dspOpen` routine, you must provide the values returned by the `dspCLListen` routine for the `remoteCID`, `remoteAddress`, `sendSeq`, `sendWindow`, and `attnSendSeq` parameters.

   You can poll the state field in the CCB to determine when the connection is open. 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.

8. You can now send and receive data and attention messages over the connection, as described in "Opening and Maintaining an ADSP Connection" beginning on page 5-13. When you are ready to close the connection, you can use the `dspClose` or `dspRemove` routine, both of which are also described in the section "Creating and Using a Connection Control Block."

9. When you are finished using the connection listener, you can use the `dspCLRemove` routine to eliminate it. Once you have called the `dspCLRemove` routine, you can release the memory you allocated for the connection listener's CCB.

Listing 5-2 illustrates the use of ADSP to establish and use a connection listener. It opens the .MPP and .DSP drivers and allocates memory for the CCB. Then it uses the `dspCLInit` routine to establish a connection listener, uses NBP to register the name of the connection end on the internet, and uses the `dspCLListen` routine to wait for a connection request. When the routine receives a connection request, it calls the `dspOpen` routine to complete the connection.

Listing 5-2 Using ADSP to establish and use a connection listener

```
VAR
   dspCCBPtr:     TPCCB;
   myDSPPBPtr:    DSPPBPtr;
   myMPPPBPtr:    MPPPBPtr;
   myNTEName:     NamesTableEntry;
   drvrRefNum:    Integer;
   mppRefNum:     Integer;
   connRefNum:    Integer;
   myErr:         OSErr;

BEGIN
   myErr := OpenDriver('.MPP', mppRefNum);
                              {open .MPP driver}
   IF myErr <> noErr THEN DoErr(myErr);
                              {check and handle error}
   myErr := OpenDriver('.DSP', drvrRefNum);
                              {open .DSP driver}
   IF myErr <> noErr THEN DoErr(myErr);
                              {check and handle error}
   {Allocate memory for data buffers.}
   dspCCBPtr := TPCCB(NewPtr(SizeOf(TRCCB)));
   myDSPPBPtr := DSPPBPtr(NewPtr(SizeOf(DSPParamBlock)));
   myMPPPBPtr := MPPPBPtr(NewPtr(SizeOf(MPPParamBlock)));
   WITH myDSPPBPtr^ DO        {set up dspCLInit parameters}
   BEGIN
      ioCRefNum := drvrRefNum;   {ADSP driver ref num}
      csCode := dspCLInit;
      ccbPtr := dspCCBPtr;       {pointer to CCB}
      localSocket := 0;          {local socket number}
   END;
   myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                              {establish a connection listener}
   IF myErr <> noErr THEN DoErr(myErr);
                              {check and handle error}
   connRefNum := myDSPPBPtr^.ccbRefNum;
                              {save CCB ref num for later}
```

```
NBPSetNTE(@myNTEName, 'The Object', 'The Type',
          '*', myDSPPBPtr^.localSocket);
                              {set up NBP names table entry}
WITH myMPPPBPtr^ DO           {set up PRegisterName parameters}
BEGIN
   interval := 7;             {retransmit every 7*8=56 ticks }
   count := 3;                { and retry 3 times}
   entityPtr := @myNTEname;   {name to register}
   verifyFlag := 1;           {verify this name}
END;
myErr := PRegisterName(myMPPPBPtr, FALSE);
                              {register this name}
IF myErr <> noErr THEN DoErr(myErr);
                              {check and handle error}

WITH myDSPPBPtr^ DO           {set up dspCLListen parameters}
BEGIN
   ioCRefNum := drvrRefNum;   {ADSP driver ref num}
   csCode := dspCLListen;
   ccbRefNum := connRefNum;   {connection ref num}
   filterAddress := AddrBlock(0);
                              {connect with anybody}
END;
myErr := PBControl(ParmBlkPtr(myDSPPBPtr), TRUE);
                              {listen for connection requests}
WHILE myDSPPBPtr^.ioResult = 1 DO
BEGIN
{Return control to user while waiting for a connection }
{ request.}
   GoDoSomething;
END;
IF myErr <> noErr THEN DoErr(myErr);
                              {check and handle error}

WITH myDSPPBPtr^ DO           {set up dspInit parameters}
BEGIN
   ioCRefNum := drvrRefNum;   {ADSP driver ref num}
   csCode := dspInit;
   ccbPtr := @dspCCB;         {pointer to CCB}
   userRoutine := @myConnectionEvtUserRoutine;
   sendQSize := qSize;        {size of send queue}
   sendQueue := dspSendQPtr;  {send-queue buffer}
   recvQSize := qSize;        {size of receive queue}
   recvQueue := dspRecvQPtr;  {receive-queue buffer}
```

```
      attnPtr := dspAttnBufPtr;  {receive-attention buffer}
      localSocket := 0;          {let ADSP assign socket}
   END;

   dspCCB.myA5 := SetCurrentA5;  {save A5 for the user routine}

   {Establish a connection end.}
   myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
   IF myErr <> noErr THEN DoErr(myErr);
                                 {check and handle error}
   connRefNum := myDSPPBPtr^.ccbRefNum;
                                 {save CCB ref num for later}

   {You received a connection request: now open a connection. }
   { The dspCLListen call has returned values into the }
   { remoteCID, remoteAddress, sendSeq, sendWindow, }
   { and attnSendSeq fields of the parameter block.}

   WITH myDSPPBPtr^ DO           {set up dspOpen parameters}
   BEGIN
      ioCRefNum := drvrRefNum;   {ADSP driver ref num}
      csCode := dspOpen;
      ccbRefNum := connRefNum;   {connection ref num}
      ocMode := ocAccept;        {open connection mode}
      ocInterval := 0;           {use default retry interval}
      ocMaximum := 0;            {use default retry maximum}
   END;
   myErr := PBControl(ParmBlkPtr(myDSPPBPtr), FALSE);
                                 {open a connection}
   IF myErr <> noErr THEN DoErr(myErr)
                                 {check and handle error}
END;     {MyCLADSP}
```

Writing a User Routine for Connection Events

When you execute the `dspInit routine`, you can specify a pointer to a routine that you provide (referred to as the *user routine*). Whenever an unsolicited connection event occurs, ADSP sets a flag in the CCB and calls the user routine. The user routine must clear the flag to acknowledge that it has read the flag field, and then it can respond to the event in any manner you deem appropriate. The CCB flags are described in“The ADSP Connection Control Block Record” beginning on page 5-35. The four following types of unsolicited connection events set flags in the CCB:

- ADSP has been informed by the remote connection end that the remote connection end is about to close the connection. An appropriate response might be to store a flag indicating that the connection end is about to close. When your application regains control, it can then display a dialog box informing the user of this event and asking whether the application should attempt to reconnect later.

- ADSP has determined that the remote connection end is not responding and so has closed the connection. Your user routine can attempt to open a new connection immediately. Alternatively, you can store a flag indicating that the connection has closed, and when your application regains control, it can display a dialog box asking the user whether to attempt to reconnect.
- ADSP has received an attention message from the remote connection end. Depending on what you are using the attention-message mechanism for, you might want to read the attention code in the `attnCode` field of the CCB and the attention message pointed to by the `attnPtr` field of the CCB.
- ADSP has received a forward reset command from the remote client end. It has then discarded all ADSP data not yet delivered, including the data in the receive queue of the local client end, and has resynchronized the connection. Your response to this event depends on the purpose for which you are using the forward reset mechanism. You might want to resend the last data you have sent or inform the user of the event.

When ADSP calls your user routine, the CPU is in interrupt-processing mode and register A1 contains a pointer to the CCB of the connection end that generated the event. You can examine the `userFlags` field of the CCB to determine what event caused the interrupt, and you can examine the `state` field of the CCB to determine the current state of the connection.

Because the CPU is set to interrupt-processing mode, your user routine must preserve all registers other than A0, A1, D0, D1, and D2. Your routine must not make any direct or indirect calls to the Memory Manager, and it cannot depend on handles to unlocked blocks being valid. If you want to use any of your application's global variables, you must save the contents of the A5 register before using the variables, and you must restore the A5 register before your routine terminates. Listing 5-1 and Listing 5-3 illustrate the use of the CCB to store the pointer to your application's global variables.

If you want to execute a routine each time an unsolicited connection event occurs but the interrupt environment is too restrictive, you can specify a `NIL` pointer to the user routine and periodically poll the `userFlags` field of the CCB.

▲ **WARNING**

When an unsolicited connection event occurs, you must clear the bit in the `userFlags` field by setting it to 0 or the connection will hang. To ensure that you do not lose any attention messages, you must read any attention messages into an internal buffer before you clear the bit in the `userFlags` field. ▲

Listing 5-3 on page 5-28 shows the user routine called by Listing 5-1 on page 5-17. When this routine is called, it first checks the CCB to determine the source of the interrupt and then clears the bit in the `userFlags` field of the CCB. If the routine has received an attention message, the user routine reads the message into an internal buffer before it clears the `flag` bit. The definitions of procedures `PushA5`, `GetMyTRCCBA5`, and `PopA5` are shown in Listing 5-3 for your convenience. In a complete application these procedures would be defined in the calling routine (see Listing 5-1 for an example).

Listing 5-3 An ADSP user routine

```
PROCEDURE PushA5;            {moves current value of A5 onto stack}
   INLINE $2F0D;             {MOVE.L A5,-(SP)}

PROCEDURE GetMyTRCCBA5;      {retrieves A5 from the head of the TRCCB }
                             { (pointed to by A1) and puts it in A5 register}
   INLINE $2A69, $FFFC;      {MOVE.L -4(A1), A5}

PROCEDURE PopA5;             {restores A5 from stack}
   INLINE $2A5F;             {MOVE.L (SP)+, A5}

PROCEDURE MyConnectionEvtUserRoutine;

BEGIN
{The connection received an unexpected connection event. Find }
{ out what kind and process accordingly.}

   PushA5;                   {save the current A5}
   GetMyTRCCBA5;             {set up A5 to point to your }
                             { application's global variables}

   WITH dspCCB.u DO
   BEGIN
      IF BAND(userFlags, eClosed) <> 0 THEN TellUserItsClosed;
      IF BAND(userFlags, eTearDown) <> 0 THEN TellUserItsBroken;
      IF BAND(userFlags, eFwdReset) <> 0 THEN TellUserItsReset;
      IF BAND(userFlags, eAttention) <> 0 THEN
      BEGIN                  {the event is an attention message}
         myAttnCode := AttnCode;
                             {get the attention code}
         CopyAttnMsg(AttnPtr, AttnSize, @myAttnData);
                             {copy the attention message into your buffer}
         tempFlag := userFlags;
         tempCFlag := eAttention;
         BClr(LongInt(tempFlag), tempCFlag);
                             {clear the flag}
         userFlags := tempFlag;
         {Do something with the message.}
      END;
      gReceivedAnEvent := TRUE
   END;
   PopA5                     {restore the current A5}
END;
```

Using ASDSP

You can write an application that uses the AppleTalk Secure Data Stream Protocol (ASDSP) to

- open a secure ASDSP connection (`sdspOpen`)
- transmit encrypted data across a secure session (`dspWrite` using the encrypt flag)
- read data decrypted by ASDSP that was sent as encrypted across a secure session (`dspRead`)

The initiator end of your ASDSP client application must call the AOCE Authentication Manager to obtain credentials to pass on to ASDSP. ASDSP passes these credentials to the recipient end of the client application and uses them to establish a secure session in which the users of the client applications at both ends of the connection are positively identified. See “About ASDSP” beginning on page 5-9 for more information about this process. ASDSP client applications at either end of a connection can send data to each other that ASDSP encrypts for transmission and then decrypts before delivering it to the client at the receiving end.

An application that currently uses ADSP needs little modification to use ASDSP. To open an ASDSP connection, the client application at each end must issue the secure data stream protocol open routine (`sdspOpen`) instead of the standard open routine (`dspOpen`). The `sdspOpen` routine uses a parameter block that, in addition to the standard ADSP parameters required to open a connection, contains the identity and credentials used in the challenge process; only the initiator end of the connection passes the credentials to ASDSP as input parameter values. The initiator and the recipient ends of a session each open the connection in a different manner:

- The initiator end of a session calls the `sdspOpen` routine using the request mode to direct ASDSP to open a connection with a specific socket.
- The recipient end of a session calls the `sdspOpen` routine in either passive mode or accept mode. A recipient end of a connection can be either of the following:
  - a specific socket that waits passively to receive an ASDSP connection request (the connection end associated with the socket calls the `sdspOpen` routine with a value of `ocPassive` for the `ocMode` parameter)
  - a connection listener that listens for connection requests and passes them on to a connection server (the connection listener calls the `sdspOpen` routine with a value of `ocAccept` for the `ocMode` parameter, and the connection server accepts and acknowledges receipt of a connection request)

You issue the `sdspOpen` routine by calling the Device Manager’s `PBControl` function and passing it a pointer to the DSP parameter block for ASDSP that holds all of the input and output parameters for the call. The parameters that the `sdspOpen` call requires differ for the initiator and recipient ends of a connection. The next section describes how to open an ASDSP connection and how to send encrypted data across it.

Opening a Secure Connection

To open a secure ASDSP connection, both the initiator and the recipient must call the `sdspOpen` routine after calling the `dspInit` routine and, optionally, the `dspOptions` routine. First this section describes how the initiator part of an application opens a secure connection. Then it describes how the recipient end of an application opens a secure connection.

From the Initiator's End

An initiator can send a request to open a secure session to

- a specific socket whose client application has opened a connection end to wait passively for a connection request
- a connection listener whose function is to accept requests for secure connections and pass those requests on to a connection server

The initiator makes either an AOCE `AuthTradeProxyForCredentials` call or an AOCE `AuthGetCredentials` call to the authentication server. It passes to the authentication server its own name and the name of the recipient and gets back the session key and the credentials for the session. For an explanation of the calls that the initiator must make to the Authentication Manager, see the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Programming Interfaces.*

Through the `sdspOpen` call, the initiator passes the credentials to ASDSP to send to the recipient. ASDSP decrypts the credentials and passes the decrypted credential information to the recipient.

To open a secure ASDSP connection, the initiator performs the following procedure:

1. Determine if the Apple Open Collaboration Environment (AOCE) software is installed by calling the `Gestalt` function. See the chapter "Introduction to AOCE" in *Inside Macintosh: AOCE Application Programming Interfaces* for a description of the selector values that you use.
2. Allocate memory for the required data structures identified in this step. The memory belongs to ASDSP until the routine completes execution, after which you can either release or reuse the memory. You must either allocate nonrelocatable memory or lock the memory until the routine completes. See the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Programming Interfaces* for a description of the memory that you need to allocate for calls that you make to that interface. The data structures that you need to allocate memory for are listed here:
   - ☐ An ASDSP parameter block of type `SDSPParamBlock`. You pass a pointer to this parameter block as the value of the `paramBlock` parameter to the `PBControl` function. (See "The ASDSP Parameter Block" on page 5-41.)
   - ☐ A workspace buffer that the `sdspOpen` routine uses internally whose size is equal to `sdspWorkSize`. The memory for this buffer must be aligned on an even boundary. You pass a pointer to this buffer as the value of the `workspace` parameter.
   - ☐ A buffer for the credentials retrieved from the authentication server and passed to ASDSP.
   - ☐ A buffer for the session key retrieved from the authentication server and passed to ASDSP. This is a data structure of type `AuthKey`.

3. Call the Authentication Manager's `AuthGetUTCTime` function to get the universal coordinated time (UTC). You base the credentials expiration time that you specify as input to the `AuthGetCredentials` function on the UTC. See the chapter "Authentication Manager" for a description of the `AuthGetUTCTime` function.
4. Obtain your (the initiator's) identity and the recipient's record ID. (You can use the local identity or get a specific identity for the initiator.) You need to pass these values to the authentication server to get the session key and credential block from the server. See the chapter "Authentication Manager" for a discussion of identities and complete instruction on how to get these values.
5. Call the Authentication Manager's `AuthGetCredentials` function or `AuthTradeProxyForCredentials` function to get the credentials and the session key. You use these values as input to the `sdspOpen` routine. See the chapter "Authentication Manager" for information on the `AuthGetCredentials` and `AuthTradeProxyForCredentials` functions.

   You pass the `AuthGetCredentials` function or `AuthTradeProxyForCredentials` function the following values returned from the functions that you called in the previous steps:

   - The initiator's identity.
   - A pointer to a buffer containing the record ID for the recipient.
   - The desired expiration time of the credentials. You use the `expiry` parameter to specify for how long you want the credentials to be valid. Credentials are valid for at most eight hours after they are returned to the initiator by the server. You base the expiration time on the UTC time returned by the `AuthGetUTCTime` function.
   - The expected length of the credentials. A buffer three times the size of a packed record ID is usually sufficient for credentials. The AOCE constant `kPackedRecordIDMaxBytes` specifies the size of a single packed record ID.
6. Call the `sdspOpen` routine to open a secure connection. To call the `sdspOpen` routine, you call the Device Manager's `PBControl` function and specify `sdspOpen` as the value of the `csCode` parameter. The parameter block for the `sdspOpen` routine includes fields also used for the standard `dspOpen` routine. In addition to these parameters, you specify parameters used in the authentication process to establish the secure connection.

   The initiator application calls the `sdspOpen` routine with a value of `ocRequest` for the `ocMode` parameter to direct ASDSP to open a connection with a specific socket on the AppleTalk internet. When you execute the `sdspOpen` routine in the `ocRequest` mode, ASDSP 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 select 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 `sdspOpen` routine.

   To use the `ocRequest` mode, you must know the complete internet address of the remote socket, and the ASDSP client at that address must either be a connection listener or have executed the `sdspOpen` routine in the `ocPassive` mode. You can use the NBP routines to obtain a list of the names of objects on the internet and to determine the internet address of a socket when you know its name. See the chapter "Name-Binding Protocol (NBP)" in this book for information on the NBP routines.

In addition to the standard ADSP parameters required for a `dspOpen` call, the initiator supplies the following input values to the `sdspOpen` call:

Parameter	Value
`secure`	To open a secure authenticated connection, pass a value of `TRUE`. To open a normal, unauthenticated connection, pass a value of `FALSE`.
`sessionKey`	A pointer to the encryption key returned from the `AuthGetCredentials` or `AuthTradeProxyForCredentials` function.
`credentialsSize`	The value that the `AuthGetCredentials` function or the `AuthTradeProxyForCredentials` function returned that specifies the length of the credentials.
`credentials`	A pointer to the credentials that the `AuthGetCredentials` function or the `AuthTradeProxyForCredentials` function returned.
`workspace`	A pointer to the workspace buffer that you allocated, which is for ASDSP's internal use.

From the Recipient End

To open a secure ASDSP connection, the recipient performs the following procedure:

1. Allocate memory for the following data structures. The memory belongs to ASDSP until the routine completes execution, after which you can either release or reuse the memory. You must either allocate nonrelocatable memory or lock the memory until the routine completes.
   - □ An ASDSP secure parameter block of type `SDSPParamBlock`. You pass a pointer to this parameter block as the value of the `paramBlock` parameter to the `PBControl` function. (See "The ASDSP Parameter Block" beginning on page 5-41.)
   - □ A workspace buffer that the `sdspOpen` routine uses internally whose size is equal to `sdspWorkSize`. The memory for this buffer must be aligned on an even boundary. You must pass a pointer to the buffer as the value of the `workspace` parameter.
   - □ A data structure of type `AuthKey` for the session key retrieved from the authentication server and passed to ASDSP. ASDSP breaks out from the credentials block the session key encrypted in the recipient's private key and returns the session key to the recipient in the `sessionKey` buffer.
   - □ A buffer for the record ID of the initiator that ASDSP returns to the recipient in response to the recipient's `sdspOpen` routine. You pass a pointer to this buffer as the value of the `initiator` parameter. ASDSP breaks out the initiator's record ID from the credential block that the initiator passes to ASDSP and returns it to the recipient. See the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Programming Interfaces* for a description of how to create a maximum-size record ID structure that is large enough to hold any record ID.
   - □ A buffer for the record ID of the intermediary that ASDSP returns to the recipient if an intermediary is found in the credentials. You pass a pointer to this buffer as the value of the `intermediary` parameter. An intermediary is a proxy that has used the `AuthTradeProxyForCredentials` function to obtain the credentials used in

the authentication process. See the chapter “Authentication Manager” in *Inside Macintosh: AOCE Application Programming Interfaces* for a discussion of the use of an intermediary and the `AuthTradeProxyForCredentials` function and for a description of how to create a maximum-size record ID structure that is large enough to hold any record ID.

2. Call the `sdspOpen` routine to open a secure connection. To call the `sdspOpen` routine, you call the Device Manager’s `PBControl` function and specify `sdspOpen` as the value of the `csCode` parameter. The parameter block for the `sdspOpen` routine includes fields also used for the standard `dspOpen` routine. In addition to these parameters, you specify parameters used in the authentication process to establish the secure connection.

   A recipient end of a connection can be either a connection listener that listens for connection requests and passes them on to a connection server or a socket that waits passively to receive a connection request.

   If the recipient is a connection listener, it calls the `sdspOpen` routine with a value of `ocAccept` for the `ocMode` parameter. The connection server accepts and acknowledges receipt of a connection request. When you call the `sdspOpen` routine, you must provide the values returned by the `dspCLListen` routine for the `remoteCID`, `remoteAddress`, `sendSeq`, `sendWindow`, and `attnSendSeq` parameters. You can poll the `state` field in the CCB to determine when the connection is open. Alternatively, you can check the result code for the `sdspOpen` routine when the routine completes execution. If the routine returns the `noErr` result code, then the connection is open.

   If the recipient is a connection end associated with a passive socket that calls the `sdspOpen` routine with a value of `ocPassive` for the `ocMode` parameter, 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.

   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 the section “The ADSP Connection Control Block Record” beginning on page 5-35 for a description of the CCB fields. 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.

   In addition to the standard ADSP parameters required for a `dspOpen` call, the recipient supplies the following input values to the `sdspOpen` call:

Parameter	Value
`sessionKey`	A pointer to a data structure of type `AuthKey`, which you allocated. ASDSP copies the session key into this buffer if an authenticated connection was successfully opened.
`workspace`	A pointer to the workspace buffer that you allocated, which is for ASDSP’s internal use.
`recipient`	The identity of the recipient.

continued

Parameter	Value
`initiator`	A pointer to a maximum-size record ID. ASDSP copies the initiator's record ID into this structure if an authenticated connection was successfully opened.
`intermediary`	A pointer to a maximum-size record ID. ASDSP copies the intermediary's record ID into this structure if an authenticated connection was successfully opened and an intermediary was used to obtain the credentials used to authenticate the call.

If a secure connection was successfully opened, ASDSP returns the following values:

Parameter	Value
`issueTime`	The time when the credentials were issued. ASDSP copies this value from the credentials.
`expiry`	The time when the credentials expire. ASDSP copies this value from the credentials.
`sessionKey`	The encryption key for the session. ASDSP copies this value from the credentials.
`initiator`	A pointer to a maximum-size record ID structure. If an authenticated connection was successfully opened, this structure holds the initiator's record ID.
`hasIntermediary`	A flag that is set to `TRUE` if an intermediary was used to obtain the credentials.
`intermediary`	A pointer to a maximum-size record ID. If an authentication connection was successfully opened and an intermediary was used to obtain the credentials, this structure holds the intermediary's record ID.

Sending Encrypted Data Across a Secure Connection

After a secure connection is established, both ends can send encrypted data over the session. ASDSP client applications use the `dspWrite` routine to send data, encrypted or not, over a secure connection. You can turn the encryption feature on or off on a message-by-message basis by setting one flag to direct ASDSP to encrypt the data and setting another flag to terminate the message.

To set these flags, you use the bits of the end-of-message (`eom`) field; this field is part of the `ioParams` variant record of the DSP parameter block that you pass to the `dspWrite` routine. For secure connections, the `eom` field comprises these two single-bit flags instead of a zero-nonzero byte. You can use the `dspEncryptMask` and `dspEOMMask` masks to set these flags, or you can use the `dspEncryptBit` or `dspEOMBit` constant.

Note
Apart from the `dspWrite` routine's `eom` parameter, the interface to ADSP remains unchanged in regard to encryption. ◆

The encryption process is transparent to the client application that receives the data; ASDSP determines if the received information is encrypted, and, if so, it decrypts the byte stream before copying the data to the read buffer specified by the `dspRead` routine.

To write data that ASDSP encrypts and then transmits or to terminate data encryption, you call the `dspWrite` routine using the Device Manager's `PBControl` function.

- Set the encrypt bit of the `eom` field (bit 1) of the DSP parameter block. To set the encrypt bit, you use the `dspEncryptMask` mask or the `dspEncryptBit` constant. Note that ASDSP checks this flag on the first write of the connection or the first write following a write for which the end-of-message flag (bit 0 of the `eom` field) is set.
- Set the end-of-message bit (bit 0) of the `eom` field to terminate the encrypted message. To set the end-of-message bit, you use the `dspEOMMask` mask or the `dspEOMBit` constant.

If you want to encrypt all messages, you can simply set the encrypt bit on all `dspWrite` calls.

ADSP Reference

This section describes the data structures and routines that are specific to ADSP and to its secure version, ASDSP. The "Data Structures" section shows the Pascal data structures for

- the ADSP connection control block
- the address block record
- the DSP parameter block
- the ASDSP version of the DSP parameter block
- the `TRSecureParams` record

The "Routines" section describes routines for setting up and tearing down an ADSP or an ASDSP (secure) connection, setting up and tearing down an ADSP connection listener, and maintaining an ADSP connection over which to send and receive data and enable encryption of the data to be sent.

Data Structures

This section describes the connection control block that you allocate for use by ADSP in maintaining the state of a connection end and the DSP parameter block that you use to specify input parameters for and receive output parameters from an ADSP routine. It also describes the address block record that you use to specify the remote connection end's AppleTalk internet address.

The ADSP Connection Control Block Record

The connection control block (CCB) data structure is a record of type `TRCCB` that consists of 242 bytes. ADSP uses the CCB to store state information about the connection end. You allocate a nonrelocatable block of memory for this data structure when you create a connection end. You may read the fields in the CCB to obtain information about the connection end, but you are not allowed to write to any of the fields except one, the `userFlags` field.

```
TYPE TRCCB =
PACKED RECORD
   ccbLink:       TPCCB;      {link to next CCB}
   refNum:        Integer;    {reference number}
   state:         Integer;    {state of the connection end}
   userFlags:     Byte;       {user flags for connection}
   localSocket:   Byte;       {local socket number}
   remoteAddress: AddrBlock;  {remote end internet address}
   attnCode:      Integer;    {attention code received}
   attnSize:      Integer;    {size of attention data}
   attnPtr:       Ptr;        {pointer to attention data}
   reserved:      PACKED ARRAY[1..220] OF Byte;
                              {reserved for use by ADSP}
END;
```

Field descriptions

`ccbLink`	A pointer to the next CCB. This field is for use by ADSP only.
`refNum`	The reference number of the CCB. This number is assigned by ADSP when you establish the connection end.
`state`	The state of the connection end, as follows:

State	Value	Meaning
`sListening`	1	The socket is a connection listener—that is, a socket that accepts ADSP requests to open connections and passes them on to a socket client. A connection listening socket passes the open-connection request on to a routine that can establish the connection on any socket. The connection listening state is ordinarily used only by connection servers.
`sPassive`	2	The socket client is inactive but capable of accepting an ADSP request to open a connection. Unlike a connection listening socket, a socket client in the `sPassive` state can accept an open-connection request only to establish itself as a connection end.
`sOpening`	3	The socket client has sent an open-connection request and is waiting for acknowledgment.
`sOpen`	4	The connection is open.

State	Value	Meaning
`sClosing`	5	The socket client has requested that ADSP close the connection, and ADSP is sending data or waiting for acknowledgment of data it has sent before closing the connection.
`sClosed`	6	The connection is closed.

`userFlags` Flags that indicate an unsolicited connection event has occurred. An unsolicited connection event is an event initiated by ADSP or the remote connection end that is not in response to any ADSP routine that you executed.

Each time an unsolicited connection event occurs, ADSP sets a flag in the `userFlags` field of the CCB and calls the routine you specified in the `userRoutine` parameter to the `dspInit` routine (if any). The user routine must read the `userFlags` field and then clear the flag to 0. ADSP cannot notify your routine of future events unless you clear the flag after each event.

ADSP recognizes four types of unsolicited connection events, one corresponding to each of the flags in this field. The events and flags are defined as follows, where bit 7 is the most significant bit:

Event	Flag bit	Meaning
`eClosed`	7	ADSP has been informed by the remote connection end that the remote connection end has closed the connection.
`eTearDown`	6	ADSP has determined that the remote connection end is not responding and so has closed the connection.
`eAttention`	5	ADSP has received an attention message from the remote connection end.
`eFwdReset`	4	ADSP has received a forward reset command from the remote connection end, has discarded all ADSP data not yet delivered—including the data in the local client end’s receive queue—and has resynchronized the connection.
None	3–0	Reserved.

`localSocket` The socket number through which DDP transmits and receives the ADSP packets.

`remoteAddress` The AppleTalk internet address of the socket used by the remote connection end.

`attnCode`	The attention code received by ADSP when the remote connection end sends an attention message.
`attnSize`	The size of the attention message received by ADSP when the remote connection end sends an attention message.
`attnPtr`	A pointer to a buffer containing the attention message received by ADSP from the remote connection end.
`reserved`	A data buffer reserved for use by ADSP.

The Address Block Record

The address block record defines a data structure of `AddrBlock` type. ADSP routines use this data type to specify the AppleTalk internet socket address of the remote connection end in the CCB. You can use NBP to get the address of an application that is registered with NBP. See the chapter "Name-Binding Protocol (NBP)" in this book for more information. ATP functions also use this data type to specify AppleTalk internet socket addresses.

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

Field descriptions

`aNet`	The network number to which the node belongs that is running the ADSP or ATP client application whose address you are specifying.
`aNode`	The node ID of the machine running the ADSP or ATP client application whose address you are specifying.
`aSocket`	The number of the socket used for the ADSP or ATP client application.

The DSP Parameter Block

The ADSP routines, which you execute by calling the Device Manager's `PBControl` function, require a pointer to a DSP parameter block that holds all of the input and output values associated with the routine. The DSP parameter block contains variant records used by particular routines. The `DSPParamBlock` data type defines the DSP parameter block.

This section defines the fields that are common to all ADSP routines that use the DSP parameter block. The fields that are used for specific routines only are defined in the descriptions of the routines to which they apply. The reserved fields, which are used internally by the .DSP driver or not at all, are not defined.

```
TYPE DSPParamBlock =
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}
   ioCRefNum:        Integer;    {driver reference number}
   csCode:           Integer;    {primary command code}
   qStatus:          LongInt;    {reserved}
   ccbRefNum:        Integer;    {CCB reference number}
CASE Integer OF
dspInit, dspCLInit:
   (ccbPtr:       TPCCB;         {pointer to CCB}
   userRoutine:   ProcPtr;       {pointer to user routine}
   sendQSize:     Integer;       {size of send queue}
   sendQueue:     Ptr;           {pointer to send queue}
   recvQSize:     Integer;       {size of receive queue}
   recvQueue:     Ptr;           {pointer to receive queue}
   attnPtr:       Ptr;           {pointer to attention-message }
                                 { buffer}
   localSocket:   Byte;          {local socket number}
   filler1:       Byte);         {filler for proper alignment}
dspOpen, dspCLListen, dspCLDeny:
   (localCID:     Integer;       {local connection ID}
   remoteCID:     Integer;       {remote connection ID}
   remoteAddress: AddrBlock;     {remote internet address}
   filterAddress: AddrBlock;     {address filter}
   sendSeq:       LongInt;       {send sequence number}
   sendWindow:    Integer;       {size of remote buffer}
   recvSeq:       LongInt;       {receive sequence number}
   attnSendSeq:   LongInt;       {attention send seq number}
   attnRecvSeq:   LongInt;       {attention receive seq num}
   ocMode:        Byte;          {connection-opening mode}
   ocInterval:    Byte;          {interval bet. open requests}
   ocMaximum:     Byte;          {retries of open-conn req}
   filler2:       Byte);         {filler for proper alignment}
dspClose, dspRemove:
   (abort:        Byte;          {abort send requests}
   filler3:       Byte);         {filler for proper alignment}
```

```
dspStatus:
   (statusCCB:    TPCCB;         {pointer to CCB}
   sendQPending:  Integer;       {bytes waiting in send queue}
   sendQFree:     Integer;       {available send-queue buffer}
   recvQPending:  Integer;       {bytes in receive queue}
   recvQFree:     Integer);      {avail receive-queue buffer}
dspRead, dspWrite:
   (reqCount:     Integer;       {requested number of bytes}
   actCount:      Integer;       {actual number of bytes}
   dataPtr:       Ptr;           {pointer to data buffer}
   eom:           Byte;          {1 if end of message}
   flush:         Byte);         {1 to send data now}
dspAttention:
   (attnCode:     Integer;       {client attention code}
   attnSize:      Integer;       {size of attention data}
   attnData:      Ptr;           {pointer to attention data}
   attnInterval:  Byte;          {reserved}
   filler4:       Byte);         {filler for proper alignment}
dspOptions:
   (sendBlocking: Integer;       {send-blocking threshold}
   sendTimer:     Byte;          {reserved}
   rtmtTimer:     Byte;          {reserved}
   badSeqMax:     Byte;          {retransmit advice threshold}
   useCheckSum:   Byte);         {DDP checksum for packets}
dspNewCID:
   (newCID:       Integer);      {new connection ID}
END;
```

Field descriptions

`ioCompletion`	A pointer to a completion routine that you can provide; the Device Manager calls your completion routine when it completes execution of the `PBControl` function, if you execute `PBControl` asynchronously and 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.
`ioResult`	The result of the function. If you call the routine asynchronously, the Device Manager sets this field to 1 as soon as you call the routine and it changes the field to the actual result code when the routine completes execution.

`ioCRefNum`	The driver reference number that is returned by the `OpenDriver` function. You must specify this number every time you call the .DSP driver.
`csCode`	The command code for the ADSP routine to be executed. You must fill in this field before calling the `PBControl` function. You use the following constants as values for this field:

csCode command	Action
`dspInit`	Create a new connection end
`dspRemove`	Remove a connection end
`dspOpen`	Open a connection
`dspClose`	Close a connection
`dspCLInit`	Create a connection listener
`dspCLRemove`	Remove a connection listener
`dspCLListen`	Post a listener request
`dspCLDeny`	Deny an open-connection request
`dspStatus`	Get status of connection end
`dspRead`	Read data from the connection
`dspWrite`	Write data on the connection
`dspAttention`	Send an attention message
`dspOptions`	Set connection end options
`dspReset`	Forward reset the connection
`dspNewCID`	Generate a CID for a connection end

`qStatus`	This field is reserved for use by ADSP.
`ccbRefNum`	The reference number of the connection control block (CCB). ADSP returns the CCB reference number in response to the `dspInit` routine. You must specify this number as a parameter to every .DSP driver routine you call subsequently.

The ASDSP Parameter Block

To open an ASDSP connection, the client application at each end must call the Device Manager's `PBControl` function with a command code that specifies the ASDSP open routine (`sdspOpen`). This section describes the ASDSP parameter block whose pointer you pass to `PBControl` to execute the `sdspOpen` routine. The ASDSP parameter block contains fields that carry the input and output parameters associated with the function. The `SDSPParamBlock` data type defines the ASDSP parameter block.

For a description of the fields that are common to both the DSP and ASDSP parameter blocks and that are used in exactly the same way, see "The DSP Parameter Block" beginning on page 5-38. For a description of the fields that are particular to the `sdspOpen` routine, see "sdspOpen" beginning on page 5-54.

```
SDSPParamBlock =
PACKED RECORD
CASE INTEGER OF
   1: (dspParamBlock: DSPParamBlock);
   2: (qLink:        QElemPtr;         {reserved}
      qType:         Integer;          {reserved}
      ioTrap:        Integer;          {reserved}
      ioCmdAddr:     Ptr;              {reserved}
      ioCompletion:  ProcPtr;          {pointer to completion routine}
      ioResult:      OSErr;            {routine result}
      ioNamePtr:     StringPtr;        {reserved}
      ioVRefNum:     Integer;          {reserved}
      ioCRefNum:     Integer;          {ASDSP driver refNum}
      csCode:        Integer;          {ASDSP driver control code}
      qStatus:       LongInt;          {reserved}
      ccbRefNum:     Integer;          {connection end refNum}
      secureParams:  TRSecureParams);  {dspOpenSecure}
END;

SDSPPBPtr = ^SDSPParamBlock;
```

Field descriptions

`csCode`	The command code for the ASDSP routine to be executed. You must fill in this field before calling the `PBControl` function. To call the `sdspOpen` routine to open a secure connection, you specify the constant `sdspOpen` as the value of this parameter.
`secureParams`	A record of type `TRSecureParams` that contains the additional parameters required to open a secure ASDSP session.

The TRSecureParams Record

The ASDSP parameter block is a variant parameter block that includes a field that is a record of type `TRSecureParams`, which defines the additional parameters required for an ASDSP session. This section shows the declaration for the `TRSecureParams` record. The routine description “sdspOpen” beginning on page 5-54 includes the field definitions for the `TRSecureParams` record.

The `TRSecureParams` record is defined as follows:

```
TYPE  TRSecureParams =
PACKED RECORD
   localCID:          Integer;       {local connection ID}
   remoteCID:         Integer;       {remote connection ID}
   remoteAddress:     AddrBlock;     {address of remote end}
   filterAddress:     AddrBlock;     {address filter}
   sendSeq:           Longint;       {local send sequence number}
   sendWindow:         Integer;       {send window size}
   recvSeq:            LongInt;       {receive sequence number}
   attnSendSeq:        LongInt;       {attention send sequence number}
   attnRecvSeq:        LongInt;       {attention receive sequence number}
   ocMode:             Byte;          {open connection mode}
   ocInterval:         Byte;          {open connection request retry }
                                      { interval}
   ocMaximum:          Byte;          {open connection request retry }
                                      { maximum}
   secure:             Boolean;       {for initiator, TRUE if session is
                                      { authenticated }
                                      {for recipient, TRUE if session was }
                                      { authenticated}
   sessionKey:         AuthKeyPtr;    {encryption key for session}
   credentialsSize:    LongInt;       {length of credentials}
   credentials:        Ptr;           {pointer to credentials}
   workspace:          Ptr;           {pointer to workspace for }
                                      { connection. Align on even boundary }
                                      { and length = sdspWorkSize}
   recipient:          AuthIdentity;  {identity of recipient or initiator }
                                      { if active mode}
   issueTime:          UTCTime;       {time when credentials were issued}
   expiry:             UTCTime;       {time when credentials expire}
   initiator:          RecordIDPtr;   {RecordID of initiator returned in }
                                      { buffer pointed to by this field}
   hasIntermediary:    Boolean;       {set if credentials has an }
                                      { intermediary}
   intermediary:       RecordIDPtr;   {RecordID of intermediary returned }
                                      { here}
END;
```

Routines

This section describes the ADSP and ASDSP routines that you use to

- establish and terminate an ADSP connection
- establish a secure (ASDSP) connection
- establish and terminate an ADSP connection listener
- maintain an ADSP connection, including sending and receiving data across an ADSP or ASDSP connection and enabling encryption of the data to be sent

You use the Device Manager's `PBControl` function for all of the ADSP and ASDSP routine calls.

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

`paramBlock`
: A pointer to the DSP parameter block that the `PBControl` function uses for DSP routines.

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

DESCRIPTION

All of the ADSP routines are implemented through a call to the `PBControl` function. The `PBControl` function takes a pointer to a parameter block and a Boolean value that specifies the mode in which the function is to be executed. You use the DSP parameter block for all ADSP calls.

The parameter block includes a field, `csCode`, in which you specify the routine selector for the particular routine to be executed; you must specify a value for this field. Each ADSP routine may use different fields of the DSP parameter block for parameters specific to that routine. The description of a function in this section includes the specific parameters used for that function. See the section "The DSP Parameter Block" beginning on page 5-38 for the complete DSP parameter block data structure.

An arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both:

Arrow	**Meaning**
→	Input
←	Output
↔	Both

Establishing and Terminating an ADSP Connection

You can use the routines described in this section to

- establish and initialize a connection end
- set the values for parameters that control the behavior of a connection end
- open an ADSP or ASDSP connection
- assign an identification number to a connection end
- close a connection end
- eliminate a connection end

dspInit

The `dspInit` routine establishes a connection end, that is, it assigns a specific socket for the ADSP connection end to use and initializes the variables that ADSP uses to maintain the connection. You use the `PBControl` function to call the `dspInit` routine. See “Routines” beginning on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspInit` for this function.
←	`ccbRefNum`	`Integer`	The CCB reference number.
→	`ccbPtr`	`TPCCB`	A pointer to the CCB.
→	`userRoutine`	`ProcPtr`	A pointer to a routine to call on connection events.
→	`sendQSize`	`Integer`	The size in bytes of the send queue.
→	`sendQueue`	`Ptr`	A pointer to the send queue.
→	`recvQSize`	`Ptr`	The size in bytes of the receive queue.
→	`recvQueue`	`Ptr`	A pointer to the receive queue.
→	`attnPtr`	`Ptr`	A pointer to the buffer for incoming attention messages.
↔	`localSocket`	`Byte`	The DDP socket number for this connection end.

Field descriptions

`csCode`	The routine selector, always equal to `dspInit` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number. The `dspInit` routine returns the CCB reference number for this connection end as the value of the `ccbRefNum` parameter. You must provide this number in all subsequent calls to this connection end.
`ccbPtr`	A pointer to the CCB that you allocated to be used by this connection end. The CCB is 242 bytes in size and is described in “The ADSP Connection Control Block Record” beginning on page 5-35. See also “Creating and Using a Connection Control Block” on page 5-12.
`userRoutine`	A pointer to a routine that ADSP is to call each time the connection end receives an unsolicited connection event. Specify `NIL` for this parameter if you do not want to supply a user routine. Connection events and user routines are discussed in “Writing a User Routine for Connection Events” beginning on page 5-26.
`sendQSize`	The size in bytes of the send queue. 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 the user’s keystrokes, a smaller buffer should be adequate. The constant `minDSPQueueSize` indicates the minimum queue size that you can use.
`sendQueue`	A pointer to the send queue that you allocated.

`recvQSize`	The size in bytes of the receive queue. A queue size of 600 bytes should work well for most applications. If you are using ADSP to receive a continuous flow of data, a larger data buffer improves performance. If your application is receiving a user's keystrokes, a smaller buffer should be adequate. The constant `minDSPQueueSize` indicates the minimum queue size that you can use.
`recvQueue`	A pointer to the receive queue that you allocated.
`attnPtr`	A pointer to the attention-message buffer that you allocated. The attention-message buffer must be the size of the constant `attnBufSize`.
`localSocket`	The DDP socket number of the socket that you want ADSP to use for this connection end. Specify 0 for this parameter to cause ADSP to assign the socket; in this case, ADSP returns the socket number when the `dspInit` routine completes execution.

DESCRIPTION

The `dspInit` routine creates and initializes a connection end. The `dspInit` routine does not open the connection end or establish a connection with a remote connection end; you must follow the `dspInit` routine with the `dspOpen` routine to perform those tasks.

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.

You must allocate memory for the send (`sendQueue`) and receive (`recvQSize`) queues and for a buffer (`attnPtr`) that holds incoming attention messages. You must also allocate a nonrelocatable block of memory (`ccbPtr`) for the CCB for this connection end.

SPECIAL CONSIDERATIONS

You must allocate nonrelocatable memory for the CCB, the send queue, the receive queue, and the attention-message buffer, and ensure that the memory remains locked until you explicitly remove the connection end by calling the `dspRemove` routine. Do not write any data to this memory except by calling ADSP routines.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`ddpSktErr`	–91	Error opening DDP socket
`errDSPQueueSize`	–1274	Send or receive queue is too small

dspOptions

The `dspOptions` routine allows you to set values for several parameters that affect the behavior of the local connection end. You use the `PBControl` function to call the `dspOptions` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspOptions` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`sendBlocking`	`Integer`	The send-blocking threshold.
→	`badSeqMax`	`Byte`	The threshold to send retransmit advice.
→	`useCheckSum`	`Byte`	A DDP checksum flag.

Field descriptions

`csCode` The routine selector, always equal to `dspOptions` for this routine.

`ccbRefNum` The connection control block (CCB) reference number that the `dspInit` routine returned.

`sendBlocking` The maximum number of bytes that may accumulate in the send queue before ADSP sends a packet to the remote connection end. ADSP sends a packet before the maximum number of bytes accumulates if the period specified by the send timer expires, if you execute the `dspWrite` routine with the flush parameter set to 1, or if a connection event requires that the local connection end send an acknowledgment packet to the remote connection end.

You can set the `sendBlocking` parameter to any value from 1 byte to the maximum size of a packet (572 bytes). If you set the `sendBlocking` parameter to 0, the current value for this parameter is not changed. The default value for the `sendBlocking` parameter is 16 bytes.

`badSeqMax` 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. Because a connection end does not acknowledge the receipt of a data packet received out of sequence, the retransmit timer of the remote connection end will expire eventually and the connection end will retransmit the data. The `badSeqMax` parameter allows you to cause the data to be retransmitted before the retransmit timer of the remote connection end has expired.

You can set the `badSeqMax` parameter to any value from 1 to 255. If you set the `badSeqMax` parameter to 0, the current value for this parameter is not changed. The default value for the `badSeqMax` parameter is 3.

`useCheckSum`	A flag specifying whether DDP should compute a checksum and include it in each packet that it sends to the remote connection end. Set this parameter to 1 if you want DDP to use checksums or to 0 if you do not want DDP to use checksums. The default value for `useCheckSum` is 0. ADSP cannot include a checksum in a packet that has a short DDP header—that is, a packet being sent over LocalTalk to a remote socket that is on the same cable as the local socket. Note that the `useCheckSum` parameter affects only whether ADSP includes a checksum in a packet that it is sending. If ADSP receives a packet that includes a checksum, it validates the checksum regardless of the setting of the `useCheckSum` parameter.

DESCRIPTION

The `dspOptions` routine lets you set values that determine the behavior of a connection end, such as the blocking factor, which is maximum number of bytes that should accumulate in the connection end's send queue before ADSP sends a packet to the remote connection end, the maximum number of out-of-sequence packets received by the connection end before ADSP sends a request for the missing packets, and whether or not DDP should use checksums for all the packets that it transmits. You can set the options for any established connection end, whether or not the connection end is open.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errRefNum`	–1280	Bad connection reference number

SEE ALSO

Use the `dspInit` routine, described on page 5-45, to return the connection control block (CCB) reference number.

dspOpen

The `dspOpen` routine opens a connection end. You can open a connection end in request mode, passive mode, accept mode, or establish mode. You use the `PBControl` function to call the `dspOpen` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	ioCompletion	ProcPtr	A pointer to completion routine.
←	ioResult	OSErr	The function result.
→	ioCRefNum	Integer	The driver reference number.
→	csCode	Integer	Always dspOpen for this function.
→	ccbRefNum	Integer	The CCB reference number.
←	localCID	Integer	The ID of this connection end.
↔	remoteCID	Integer	The ID of remote connection end.
↔	remoteAddress	AddrBlock	A remote internet address.
→	filterAddress	AddrBlock	A filter for open-connection requests.
↔	sendSeq	LongInt	The initial send sequence number.
↔	sendWindow	Integer	The initial size of remote receive queue.
→	recvSeq	LongInt	The initial receive sequence number.
↔	attnSendSeq	LongInt	The attention send sequence number.
→	attnRecvSeq	LongInt	The attention receive sequence number.
→	ocMode	Byte	The connection-opening mode.
→	ocInterval	Byte	The interval between open requests.
→	ocMaximum	Byte	The number of open-connection request retries.

The use of parameters by the dspOpen routine depends on the mode in which the routine is executed, as follows:

ocRequest		**ocPassive**		**ocAccept**		**ocEstablish**	
→	ioCompletion	→	ioCompletion	→	ioCompletion	→	ioCompletion
←	ioResult	←	ioResult	←	ioResult	←	ioResult
→	ioCRefNum	→	ioCRefNum	→	ioCRefNum	→	ioCRefNum
→	csCode	→	csCode	→	csCode	→	csCode
→	ccbRefNum	→	ccbRefNum	→	ccbRefNum	→	ccbRefNum
←	localCID	←	localCID	←	localCID	—	localCID
←	remoteCID	←	remoteCID	→	remoteCID	→	remoteCID
→	remoteAddress	←	remoteAddress	→	remoteAddress	→	remoteAddress
→	filterAddress	→	filterAddress	—	filterAddress	—	filterAddress
←	sendSeq	←	sendSeq	→	sendSeq	→	sendSeq
←	sendWindow	←	sendWindow	→	sendWindow	→	sendWindow
—	recvSeq	—	recvSeq	—	recvSeq	→	recvSeq
←	attnSendSeq	←	attnSendSeq	→	attnSendSeq	→	attnSendSeq
—	attnRecvSeq	—	attnRecvSeq	—	attnRecvSeq	→	attnRecvSeq
→	ocMode	→	ocMode	→	ocMode	→	ocMode
→	ocInterval	→	ocInterval	→	ocInterval	—	ocInterval
→	ocMaximum	→	ocMaximum	→	ocMaximum	—	ocMaximum

Key: → input ← output ↔ input and output — not used

Field descriptions

csCode	The routine selector, always equal to dspOpen for this routine.
ccbRefNum	The connection control block (CCB) reference number that was returned by the dspInit routine for the connection end that you want to use.
localCID	The identification number of the local connection end. This number is assigned by ADSP when you open the connection. ADSP includes this number in every packet sent to a remote connection end. Before you call the dspOpen routine in ocEstablish mode, you must call the dspNewCID routine to cause ADSP to assign this value.
remoteCID	The identification number of the remote connection end. This parameter is returned by the dspOpen routine in the ocRequest and ocPassive modes. A connection server must provide this number to the dspOpen routine when the server executes the routine in ocAccept mode; in this case, the connection server obtains the remoteCID value from the dspCLListen routine. You must provide the remoteCID value to the dspOpen routine when you use the routine in ocEstablish mode.
remoteAddress	The internet address of the remote socket with which you wish to establish communications. This address consists of a 2-byte network number, a 1-byte node ID, and a 1-byte socket number. You must provide this parameter when you call the dspOpen routine in the ocRequest or ocEstablish mode. This parameter is returned by the dspOpen routine when you call the routine in the ocPassive mode. When you call the dspOpen routine in the ocAccept mode, you must use the value for the remoteAddress parameter that was returned by the dspCLListen routine.
filterAddress	The internet address of the socket from which you will accept a connection request. The address consists of three fields: a 2-byte network number, a 1-byte node ID, and a 1-byte socket number. Specify 0 for any of these fields for which you wish to impose no restrictions. If you specify a filter address of $00082500, for example, the connection end accepts a connection request from any socket at node $25 of network $0008. Set the filterAddress parameter equal to the remoteAddress parameter to accept a connection only with the socket to which you sent a connection request. When you execute the dspOpen routine in the ocPassive mode, you can receive a connection request from any ADSP connection end on the internet. When you execute the dspOpen routine in the ocRequest mode, your connection end can receive a connection request acknowledgment from an address different from the one you specified in the remoteAddress parameter only if the remote address you specified was that of a connection listener. In either case, you can use the filterAddress parameter to avoid acknowledging unwanted connection requests. When you execute the dspOpen routine in the ocAccept mode, your connection listener has already received and decided to accept the connection request. You can specify a filter address for a

	connection listener with the `dspCLListen` routine. A connection server can use the `dspCLDeny` routine to deny a connection request that was accepted by its connection listener. You cannot use the filter address when you execute the `dspOpen` routine in `ocEstablish` mode.
`sendSeq`	The sequence number of the first byte that the local connection end will send to the remote connection end. ADSP uses this number to coordinate communications and to check for errors. ADSP returns a value for the `sendSeq` parameter when you execute the `dspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `dspOpen` routine in the `ocAccept` mode, you must specify the value for the `sendSeq` parameter that was returned by the `dspCLListen` routine. You must provide the value for this parameter when you execute the `dspOpen` routine in the `ocEstablish` mode.
`sendWindow`	The sequence number of the last byte that the remote connection end has buffer space to receive. ADSP uses this number to coordinate communications and to check for errors. ADSP returns a value for the `sendWindow` parameter when you execute the `dspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `dspOpen` routine in the `ocAccept` mode, you must specify the value for the `sendWindow` parameter that was returned by the `dspCLListen` routine. You must provide the value for this parameter when you execute the `dspOpen` routine in the `ocEstablish` mode.
`recvSeq`	The sequence number of the next byte that the local connection end expects to receive. ADSP uses this number to coordinate communications and to check for errors. You must provide the value for this parameter when you execute the `dspOpen` routine in the `ocEstablish` mode. The `dspOpen` routine does not use this parameter when you execute it in any other mode.
`attnSendSeq`	The sequence number of the next attention packet that the local connection end will transmit. ADSP uses this number to coordinate communications and to check for errors. ADSP returns a value for the `attnSendSeq` parameter when you execute the `dspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `dspOpen` routine in the `ocAccept` mode, you must specify the value for the `attnSendSeq` parameter that was returned by the `dspCLListen` routine. You must provide the value for this parameter when you execute the `dspOpen` routine in the `ocEstablish` mode.
`attnRecvSeq`	The sequence number of the next attention packet that the local connection end expects to receive. ADSP uses this number to ensure that packets are delivered in the correct order and to check for errors. You must provide a value for this parameter when you execute the `dspOpen` routine in the `ocEstablish` mode. The `dspOpen` routine does not use this parameter when you execute it in any other mode.

`ocMode` The mode in which the `dspOpen` routine is to operate, as follows:

Mode	Value	Meaning
`ocRequest`	1	ADSP attempts to open a connection with the socket you specify.
`ocPassive`	2	The connection end waits to receive a connection request.
`ocAccept`	3	The connection server accepts and acknowledges receipt of a connection request.
`ocEstablish`	4	ADSP considers the connection established and open; you are responsible for setting up and synchronizing both connection ends.

`ocInterval` The period between transmissions of open-connection requests. If the remote connection end does not acknowledge or deny an open-connection request, ADSP retransmits the request after a time period specified by this parameter. The time period used by ADSP is (`ocInterval` × 10) ticks, or (`ocInterval` / 6) seconds. For example, if you set the `ocInterval` parameter to 3, the time period between retransmissions is 30 ticks (1/2 second). You can set the `ocInterval` parameter to any value from 1 (1/6 second) to 180 (30 seconds). If you specify 0 for the `ocInterval` parameter, ADSP uses the default value of 6 (1 second).

You must provide a value for the `ocInterval` parameter when you execute the `dspOpen` routine in the `ocRequest`, `ocPassive`, or `ocAccept` mode. The `dspOpen` routine does not use this parameter when you execute it in the `ocEstablish` mode.

`ocMaximum` The maximum number of times to retransmit an open-connection request before ADSP terminates execution of the `dspOpen` routine. If you specify 0 for the `ocMaximum` parameter, ADSP uses the default value of 3. If you specify 255 for the `ocMaximum` parameter, ADSP retransmits the open-connection request indefinitely until the remote connection end either acknowledges or denies the request.

You must provide a value for the `ocMaximum` parameter when you execute the `dspOpen` routine in the `ocRequest`, `ocPassive`, or `ocAccept` mode. The `dspOpen` routine does not use this parameter when you execute it in the `ocEstablish` mode.

DESCRIPTION

The `dspOpen` routine opens a connection end. You set the `ocMode` field of the parameter block to specify the opening mode that the `dspOpen` routine is to use. The `dspOpen` routine puts a connection end into one of the four following opening modes:

- The `ocRequest` mode, in which ADSP attempts to open a connection with the socket at the internet address you specify as the `remoteAddress` parameter. If the socket you specify as a remote address is a connection listener, it is possible that your application will receive a connection acknowledgment and request from a different address than the one to which you sent the open-connection request. You can use the `filterAddress` parameter to restrict the addresses with which you will accept a connection.

The `dspOpen` routine completes execution in the `ocRequest` mode when one of the following occurs: ADSP establishes a connection, your connection end receives a connection denial from the remote connection end, your connection end denies the connection request returned by a connection listener, or ADSP cannot complete the connection within the maximum number of retries that you specified with the `ocMaximum` parameter.

- The `ocPassive mode`, in which the connection end waits to receive an open-connection request from a remote connection end. You can use the `filterAddress` parameter to restrict the addresses from which you will accept a connection request.

  The `dspOpen` routine completes execution in the `ocPassive` mode when ADSP establishes a connection or when either connection end receives a connection denial.

- The `ocAccept` mode, used by connection servers to complete an open-connection dialog. When a connection server is informed by its connection listener that the connection listener has received an open-connection request, the connection server calls the `dspInit` routine to establish a connection end and then calls the `dspOpen` routine in `ocAccept` mode to complete the connection. You must obtain the following parameters from the `dspCLListen` routine and provide them to the `dspOpen` routine: `remoteAddress`, `remoteCID`, `sendSeq`, `sendWindow`, and `attnSendSeq`. Connection listeners and connection servers are described in "Creating and Using a Connection Listener" beginning on page 5-22 and in "Establishing and Terminating an ADSP Connection" beginning on page 5-44. See "Connection Listeners" on page 5-7 for a brief introduction to connection listeners.

  The `dspOpen` routine completes execution in the `ocAccept` mode when ADSP establishes a connection or when either connection end receives a connection denial.

- The `ocEstablish` mode, in which ADSP considers the connection end established and the connection state open. This mode is for use by clients that determine their connection-opening parameters without using ADSP or the .DSP driver to do so.

  You must first use the `dspInit` routine to establish a connection end and then execute the `dspNewCID` routine to obtain an identification number (ID) for the local connection end. You must then communicate with the remote connection end to send it the local connection ID and to determine the values of the following parameters: `remoteAddress`, `remoteCID`, `sendSeq`, `sendWindow`, `recvSeq`, `attnSendSeq`, and `attnRecvSeq`. Only then can you execute the `dspOpen` routine in the `ocEstablish` mode.

  The `dspOpen` routine completes execution in the `ocEstablish` mode immediately.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errOpenDenied`	–1273	Open request denied by recipient
`errOpening`	–1277	Attempt to open connection failed
`errState`	–1278	Connection end must be closed
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number

sdspOpen

The `sdspOpen` routine opens a secure (ASDSP) connection and causes ASDSP to perform the challenge-and-reply process that authenticates the ASDSP clients at either end of the connection. You use the `PBControl` function to call the `sdspOpen` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

`ioCompletion`	`ProcPtr`	A pointer to completion routine.
`ioResult`	`OSErr`	A result code.
`ioCRefNum`	`Integer`	The ADSP driver reference number.
`csCode`	`Integer`	Always `sdspOpen` for this function.
`ccbRefNum`	`Integer`	The CCB reference number for connection end.
`localCID`	`Integer`	The ID of this connection end.
`remoteCID`	`Integer`	The ID of remote connection end.
`remoteAddress`	`AddrBlock`	A remote internet address.
`filterAddress`	`AddrBlock`	A filter for open connection end.
`sendSeq`	`LongInt`	The initial send sequence number.
`sendWindow`	`Integer`	The initial size of remote receive queue.
`recvSeq`	`LongInt`	Not used for ASDSP.
`attnSendSeq`	`LongInt`	The attention send sequence number.
`attnRecvSeq`	`LongInt`	Not used for ASDSP.
`ocMode`	`Byte`	The connection-opening mode.
`ocInterval`	`Byte`	The interval between open requests.
`ocMaximum`	`Byte`	The maximum number of retries of the open-connection request.
`secure`	`Boolean`	A flag that determines if ASDSP authenticates the connection.
`sessionKey`	`AuthKeyPtr`	A pointer to the session encryption key.
`credentialsSize`	`LongInt`	The length of credentials.
`credentials`	`Ptr`	A pointer to credentials.
`workspace`	`Ptr`	A pointer to workspace for connection.
`recipient`	`AuthIdentity`	The identity of recipient.
`issueTime`	`UTCTime`	The time when credentials were issued.
`expiry`	`UTCTime`	The time when credentials expire.
`initiator`	`RecordIDPtr`	A pointer to record ID of initiator.
`hasIntermediary`	`Boolean`	`TRUE` if credentials has an intermediary.
`intermediary`	`RecordIDPtr`	A pointer to record ID of intermediary.

The use of parameters by the `sdspOpen` routine depends on the mode in which the routine is executed, as follows:

ocRequest	**ocPassive**	**ocAccept**
→ `ioCompletion`	→ `ioCompletion`	→ `ioCompletion`
← `ioResult`	← `ioResult`	← `ioResult`
→ `ioCRefNum`	→ `ioCRefNum`	→ `ioCRefNum`
→ `csCode`	→ `csCode`	→ `csCode`
→ `ccbRefNum`	→ `ccbRefNum`	→ `ccbRefNum`
← `localCID`	← `localCID`	← `localCID`
← `remoteCID`	← `remoteCID`	→ `remoteCID`
→ `remoteAddress`	← `remoteAddress`	→ `remoteAddress`
→ `filterAddress`	→ `filterAddress`	— `filterAddress`
← `sendSeq`	← `sendSeq`	→ `sendSeq`
← `sendWindow`	← `sendWindow`	→ `sendWindow`
— `recvSeq`	— `recvSeq`	— `recvSeq`
← `attnSendSeq`	← `attnSendSeq`	→ `attnSendSeq`
— `attnRecvSeq`	— `attnRecvSeq`	— `attnRecvSeq`
→ `ocMode`	→ `ocMode`	→ `ocMode`
→ `ocInterval`	→ `ocInterval`	→ `ocInterval`
→ `ocMaximum`	→ `ocMaximum`	→ `ocMaximum`
→ `secure`	← `secure`	← `secure`
→ `sessionKey`	← `sessionKey`	← `sessionKey`
→ `credentialsSize`	— `credentialsSize`	— `credentialsSize`
→ `credentials`	— `credentials`	— `credentials`
→ `workspace`	→ `workspace`	→ `workspace`
— `recipient`	→ `recipient`	→ `recipient`
— `issueTime`	← `issueTime`	← `issueTime`
— `expiry`	← `expiry`	← `expiry`
— `initiator`	↔ `initiator`	↔ `initiator`
— `hasIntermediary`	← `hasIntermediary`	← `hasIntermediary`
— `intermediary`	↔ `intermediary`	↔ `intermediary`

Key: → input ← output ↔ input and output — not used

Field descriptions

`csCode`	The routine selector, always equal to `sdspOpen` for this routine.
`ccbRefNum`	This field is used in the same way that it is used for ADSP. See the description of this field under “dspOpen” beginning on page 5-48.
`localCID`	This field is used in the same way that it is used for ADSP. See the description of this field under “dspOpen” beginning on page 5-48.

`remoteCID`	The identification number of the remote connection end. This parameter is returned by the `sdspOpen` routine in the `ocRequest` and `ocPassive` modes. A connection server must provide this number to the `sdspOpen` routine when the server executes the routine in `ocAccept` mode; in this case, the connection server obtains the `remoteCID` value from the `dspCLListen` routine.
`remoteAddress`	The internet address of the remote socket with which you wish to establish communications. This address consists of a 2-byte network number, a 1-byte node ID, and a 1-byte socket number. You must provide this parameter when you call the `sdspOpen` routine in the `ocRequest` or `ocAccept` mode. When you call the `sdspOpen` routine in the `ocAccept` mode, you must use the value for the `remoteAddress` parameter that was returned by the `dspCLListen` routine. This parameter is returned by the `sdspOpen` routine when you call the routine in the `ocPassive` mode.
`filterAddress`	This field is used in the same way that it is used for ADSP. See the description of this field under "dspOpen" beginning on page 5-48.
`sendSeq`	The sequence number of the first byte that the local connection end will send to the remote connection end. ASDSP uses this number to coordinate communications and to check for errors. ASDSP returns a value for the `sendSeq` parameter when you execute the `sdspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `sdspOpen` routine in the `ocAccept mode`, you must specify the value for the `sendSeq` parameter that was returned by the `dspCLListen` routine.
`sendWindow`	The sequence number of the last byte that the remote connection end has buffer space to receive. ASDSP uses this number to coordinate communications and to check for errors. ASDSP returns a value for the `sendWindow` parameter when you execute the `sdspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `sdspOpen` routine in the `ocAccept` mode, you must specify the value for the `sendWindow` parameter that was returned by the `dspCLListen` routine.
`recvSeq`	This field is not used by ASDSP.
`attnSendSeq`	The sequence number of the next attention packet that the local connection end will transmit. ASDSP uses this number to coordinate communications and to check for errors. ASDSP returns a value for the `attnSendSeq` parameter when you execute the `sdspOpen` routine in the `ocRequest` or `ocPassive` mode. When you execute the `sdspOpen` routine in the `ocAccept` mode, you must specify the value for the `attnSendSeq` parameter that was returned by the `dspCLListen` routine.
`attnRecvSeq`	This field is not used by ASDSP.

`ocMode`	The mode in which the `sdspOpen` routine is to operate, as follows:

Mode	Value	Meaning
`ocRequest`	1	ADSP attempts to open a connection with the remote socket you specify.
`ocPassive`	2	The connection end waits to receive a connection request.
`ocAccept`	3	The connection server accepts and acknowledges receipt of a connection request.

`ocInterval`	This field is used in the same way that it is used for ADSP. See the description of this field under "dspOpen" beginning on page 5-48.
`ocMaximum`	This field is used in the same way that it is used for ADSP. See the description of this field under "dspOpen" beginning on page 5-48.
`secure`	A flag that determines whether ASDSP authenticates the connection. On input for the initiator end, you must set this value to `TRUE` if you want ASDSP to authenticate the connection. You must provide a value for the `secure` parameter when you execute the `sdspOpen` routine in the `ocRequest` mode. ASDSP returns a value of `TRUE` for this parameter to the recipient for all modes if the session was authenticated.
`sessionkey`	A pointer to a buffer containing the session key returned by the Authentication Manager's `AuthGetCredentials` or `AuthTradeProxyForCredentials` function. The initiator connection end must provide an input value for this parameter. For the recipient connection end, ASDSP breaks out the session key from the credentials block and returns a copy of the session key as the value of this parameter. See the description of the data structures that you need to allocate for ASDSP in the section "Opening a Secure Connection" beginning on page 5-30 for more information about the buffer.
`credentialsSize`	The size in bytes of credentials returned by the Authentication Manager's `AuthTradeProxyForCredentials` or `AuthGetCredentials` function.You must provide a value for the `credentialsSize` parameter when you execute the `sdspOpen` routine in the `ocRequest` mode. This parameter is not used for the recipient end of the connection when you call the `sdspOpen` routine in `ocAccept` mode or `ocPassive` mode.
`credentials`	A pointer to the credentials for this session that the Authentication Manager's `AuthTradeProxyForCredentials` or `AuthGetCredentials` function returned when you called it. Specify the size in bytes of the credential block pointed to by this parameter as the value of the `credentialsSize` parameter when you call the `sdspOpen` routine in the `ocRequest` mode. This parameter is not used for the recipient end of the connection when you call the `sdspOpen` routine in `ocAccept` mode or `ocPassive` mode. See the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Programming Interfaces.*

`workspace`	A pointer to a buffer that you allocate as workspace for the `sdspOpen` routine's internal use. The memory for the buffer that you allocate must be aligned on an even boundary and must be equal in size to the `sdspWorkSize` constant, which is 2048 bytes.
`recipient`	When the value of the `ocMode` parameter is `ocAccept`, you specify the identity of the connection server as the value of the `recipient` parameter. When the value of the `ocMode` parameter is `ocPassive`, you specify the identity of the socket that is the recipient of the request call as the value of the `recipient` parameter. This field is not used when the `ocMode` parameter value is `ocRequest`.
`issueTime`	The time when the authentication credentials were issued. Together with the `expiry` parameter value, the `issueTime` parameter specifies the period of time for which the credentials are valid. ASDSP extracts the value for the `issueTime` parameter from the decrypted credentials. ASDSP returns this value when the mode is `ocPassive` or `ocAccept`. The `issueTime` field is not used when the `ocMode` parameter value is `ocRequest`.
`expiry`	The time when the authentication credentials expire. Together with the `issueTime` parameter value, the `expiry` parameter specifies the duration for which the credentials are valid. ASDSP extracts the value for the `expiry` parameter from the decrypted credentials. This field is not used when the `ocMode` parameter value is `ocRequest`.
`initiator`	A pointer to the record ID of the initiator that ASDSP returns when the value of the `ocMode` parameter is `ocAccept` or `ocPassive`. ASDSP extracts this value from the encrypted credentials. This field is not used when the `ocMode` parameter value is `ocRequest`.
`hasIntermediary`	A flag that ASDSP sets if the credentials have an intermediary. When this flag is set, a proxy was used; an intermediary used the `AuthTradeProxyForCredentials` function to obtain the credentials used in the authentication process. The `sdspOpen` routine returns this value when the `ocMode` parameter value is `ocPassive` or `ocAccept`.
`intermediary`	A pointer to a buffer that is used to store the record ID of the intermediary, if ASDSP finds an intermediary in the credentials. The `sdspOpen` routine returns this value when the `ocMode` parameter value is `ocPassive` or `ocAccept`.

DESCRIPTION

The `sdspOpen` routine opens a secure connection end if the identities of both the initiator and the recipient connection ends can be proven in the authentication process. You set the `ocMode` field of the parameter block to specify the opening mode that the `sdspOpen` routine is to use. The `sdspOpen` routine puts a connection end into one of the three following opening modes:

- In the `ocRequest` mode, ASDSP attempts to open a connection with the socket at the internet address you specify as the `remoteAddress` parameter.

- In the `ocPassive` mode, the connection end waits to receive an open-connection request from a remote connection end. You can use the `filterAddress` parameter to restrict the addresses from which you will accept a connection request.
- In the `ocAccept` mode, connection servers complete open-connection dialogs. When a connection server is informed by its connection listener that the connection listener has received an open-connection request, the connection server calls the `dspInit` routine to establish a connection end and then calls the `sdspOpen` routine in `ocAccept` mode to complete the connection. Connection listeners and connection servers are described in "Creating and Using a Connection Listener" beginning on page 5-22 and in "Establishing and Terminating an ADSP Connection" beginning on page 5-44. See "Connection Listeners" on page 5-7 for a brief introduction to connection listeners.

Except for the authentication process, these three modes are used by ASDSP and ADSP in the same way and their behavior is the same. See the description of how these modes are used in "dspOpen" beginning on page 5-48.

If ASDSP cannot successfully complete the authentication process, ASDSP tears down the connection and the `sdspOpen` calls made by both the initiator and the recipient return a result code reporting the reason why the authentication process failed. For the conditions that can cause the authentication process to fail, see the list of result codes that follows.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errOpenDenied`	–1273	Open request denied by recipient
`errFwdReset`	–1276	A forward reset caused ASDSP to terminate the request
`errOpening`	–1277	Attempt to open connection failed
`errState`	–1278	Connection end is not open
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number
`kOCEUnsupportedCredentialsVersion`	–1543	Credentials version not supported
`kOCEBadEncryptionMethod`	–1559	During the authentication process, the ASDSP implementations could not agree on an encryption method to be used (ASDSP can support multiple stream encryption methods. In Release 1, only RC4 and "no encryption" are supported.)
`kOCENoASDSPWorkSpace`	–1570	You passed `NIL` for the workspace parameter
`kOCEAuthenticationTrouble`	–1571	Authentication process failed

dspNewCID

The `dspNewCID` routine creates a connection ID to be used in setting up a connection. You use the `PBControl` function to call the `dspNewCID` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspNewCID` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
←	`newCID`	`Integer`	The ID of new connection.

Field descriptions

`csCode`	The routine selector, always equal to `dspNewCID` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that was returned by the `dspNewCID` routine for the connection end that you want to use.
`newCID`	The connection-end ID that this routine returns. You must provide this number to the client of the remote connection end so that it can use it for the `remoteCID` parameter when it calls the `dspOpen` routine.

DESCRIPTION

The `dspNewCID` routine causes ADSP to assign an ID to a connection end without opening the connection end or attempting to establish a connection with a remote connection end. Use this routine only if you implement your own protocol to establish communication with a remote connection end. You must first use the `dspInit` routine to establish a connection end. Next, you must call the `dspNewCID` routine to obtain a connection-end ID. Then you must establish communication with a remote connection end and pass the ID to the remote connection end. Finally, you must call the `dspOpen` routine in `ocEstablish` mode to cause ADSP to open the connection.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errState`	–1278	Connection is not closed
`errRefNum`	–1280	Bad connection reference number

SEE ALSO

To establish a connection, use the `dspInit` routine, described on page 5-45.

To obtain a connection-end ID, use the `sdspOpen` routine, described on page 5-54.

To open a connection in `ocEstablish` mode, use the `dspOpen` routine, described on see page 5-48.

dspClose

The `dspClose` routine closes a connection end. You use the `PBControl` function to call the `dspClose` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspClose` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`abort`	`Byte`	A value specifying to abort send requests if not 0.

Field descriptions

`csCode`	The routine selector, always equal to `dspClose` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that was returned by the `dspNewCID` routine for the connection end that you want to close.
`abort`	A value that specifies whether or not to send all of the data in the send queue and all outstanding messages before closing the connection end. If the abort parameter is nonzero, ADSP cancels any outstanding requests to send data packets (such as the `dspAttention` routine) and discards all data in the send queue. If the abort parameter is 0, ADSP does not close the connection end until all of the data in the send queue and all outstanding attention messages have been sent and acknowledged.

DESCRIPTION

The `dspClose` routine closes the connection end. The connection end is still established; that is, ADSP retains ownership of the CCB, send queue, receive queue, and attention-message buffer. You can continue to read bytes from the receive queue after you have called the `dspClose` routine. Use the `dspRemove` routine instead of the `dspClose` routine if you are finished with reading bytes from the receive queue and want to release the memory associated with the connection end.

SPECIAL CONSIDERATIONS

The `dspClose` routine does not return an error if you call it for a connection end that is already closed.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errRefNum`	–1280	Bad connection reference number

SEE ALSO

For information on how to remove a connection end and release the memory associated with it, see the description of the `dspRemove` routine that follows.

dspRemove

The `dspRemove` routine closes any open connection and eliminates the connection end, releasing all memory associated with it. You use the `PBControl` function to call the `dspRemove` routine. See “Routines” on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspRemove` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`abort`	`Byte`	A value specifying to abort connection if not 0.

Field descriptions

`csCode`	The routine selector, always equal to `dspRemove` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that was returned by the `dspNewCID` routine for the connection end that you want to remove.
`abort`	A value that specifies whether or not to send all of the data in the send queue and all outstanding messages before closing the connection end. If the abort parameter is nonzero, ADSP cancels any outstanding requests to send data packets (such as the `dspAttention` routine) and discards all data in the send queue. If the abort parameter is 0, ADSP does not close the connection end until all of the data in the send queue and all outstanding attention messages have been sent and acknowledged.

DESCRIPTION

The `dspRemove` routine closes the connection end whose connection control block (CCB) you specify, and it eliminates that connection end; that is, ADSP no longer retains control of the CCB, send queue, receive queue, and attention-message buffer. You cannot continue to read bytes from the receive queue after you have called the `dspRemove` routine. After you call the `dspRemove` routine, you can release all of the memory you allocated for the connection end if you do not intend to reopen the connection end.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errRefNum`	–1280	Bad connection reference number

Establishing and Terminating an ADSP Connection Listener

A connection listener is a special kind of connection end that listens for open-connection requests from remote connection ends. Connection listeners are used by ***connection servers***—that is, programs that assign a socket for the local connection end only after they receive a connection request from a remote connection end. A single connection listener can receive connection requests from any number of remote connection ends.

You can use the routines in this section to

- establish a connection listener
- cause the connection listener to listen for a connection request
- deny a connection request
- close and eliminate a connection listener

dspCLInit

The `dspCLInit` routine establishes and initializes a connection listener. You use the `PBControl` function to call the `dspCLInit` routine. See “Routines” on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspCLInit` for this function.
←	`ccbRefNum`	`Integer`	The CCB reference number.
→	`ccbPtr`	`TPCCB`	A pointer to CCB.
↔	`localSocket`	`Byte`	The local DDP socket number.

Field descriptions

`csCode`	The routine selector, always equal to `dspCLInit` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number. The `dspCLInit` routine returns this value.You must provide this number in all subsequent `dspCLListen` and `dspCLRemove` calls to this connection listener.
`ccbPtr`	A pointer to the CCB that you allocated. The CCB is 242 bytes in size.
`localSocket`	The number of the DDP socket that you want ADSP to use for this connection end. Specify 0 for this parameter to cause ADSP to assign the socket; in this case, ADSP returns the socket number when the `dspCLInit` routine completes execution.

DESCRIPTION

The `dspCLInit` routine establishes a connection listener; that is, it assigns a specific socket for use by ADSP and initializes the variables that ADSP uses to maintain a connection listener. The `dspCLInit` routine does not cause the connection listener to listen for connection requests; you must follow the `dspCLInit` routine with the `dspCLListen` routine to activate the connection listener.

You must allocate a block of nonrelocatable memory for a CCB before you call the `dspCLInit` routine and pass a pointer to that CCB as the value of the `ccbPtr` parameter. See the section “Creating and Using a Connection Control Block” on page 5-12 and the section “The ADSP Connection Control Block Record” on page 5-35 for more information.

SPECIAL CONSIDERATIONS

The connection control block for which you allocate memory belongs to ADSP until you explicitly remove the connection listener. You cannot release the memory for the CCB until after you eliminate the connection listener.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`ddpSktErr`	–91	Error opening socket

SEE ALSO

To establish a connection end that is not a connection listener, use the `dspInit` routine described on page 5-45.

To eliminate a connection listener, use the `dspCLRemove` routine, described on page 5-68.

dspCLListen

The `dspCLListen` routine causes a connection listener to listen for connection requests. You use the `PBControl` function to call the `dspCLListen` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspCLListen` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
←	`remoteCID`	`Integer`	The ID of the remote connection end.
←	`remoteAddress`	`AddrBlock`	The remote internet address.
→	`filterAddress`	`AddrBlock`	A filter for open-connection requests.
←	`sendSeq`	`LongInt`	The initial send sequence number.
←	`sendWindow`	`Integer`	The initial size of the remote receive queue.
←	`attnSendSeq`	`LongInt`	The attention send sequence number.

Field descriptions

`csCode`	The routine selector, always `dspCLListen` for this routine.
`ccbRefNum`	The CCB reference number that the `dspCLInit` routine returned.
`remoteCID`	The identification number of the remote connection end. You must pass this value to the `dspOpen` routine when you open the connection or to the `dspCLDeny` routine when you deny the connection request. The `dspCLListen` routine returns this number.
`remoteAddress`	The internet address of the remote socket that sent a request to open a connection. This address consists of a 2-byte network number, a 1-byte node ID, and a 1-byte socket number. You must pass this value to the `dspOpen` routine when you open the connection or to the `dspCLDeny` routine when you deny the connection request.

`filterAddress`	The internet address of the socket from which you will accept a connection request. The address consists of three fields: a 2-byte network number, a 1-byte node ID, and a 1-byte socket number. Specify 0 for any of these fields for which you wish to impose no restrictions. If you specify a filter address of $00082500, for example, the connection listener accepts a connection request from any socket at node $25 of network $0008.
`sendSeq`	The sequence number of the first byte that the local connection end will send to the remote connection end. ADSP uses this number to coordinate communications and to check for errors. You must pass this value to the `dspOpen` routine when you open the connection.
`sendWindow`	The sequence number of the last byte that the remote connection end has buffer space to receive. ADSP uses this number to coordinate communications and to check for errors. You must pass this value to the `dspOpen` routine when you open the connection.
`attnSendSeq`	The sequence number of the next attention packet that the local connection end will transmit. ADSP uses this number to ensure that attention packets are delivered in the correct order and to check for errors. You must pass this value to the `dspOpe`n routine when you open the connection.

DESCRIPTION

The `dspCLListen` routine initiates the connection listener. You must have already used the `dspCLInit` routine to establish a connection listener before using the `dspCLListen` routine. The `dspCLListen` routine is used only by connection servers.

When ADSP receives an open-connection request from a socket that satisfies the address requirements of the `filterAddress` parameter, it returns values for the `remoteCID`, `remoteAddress`, `sendSeq`, `sendWindow`, and `attnSendSeq` parameters and completes execution of the `dspCLListen` routine. You must then either accept the open-connection request by calling the `dspOpen` routine in the `ocAccept` mode or deny the request by calling the `dspCLDeny` routine.

You can call the `dspCLListen` routine several times, specifying the same connection listener. For example, if you wanted to accept connections from any or all of three different addresses, you could call the `dspCLListen` routine three times with a different value for the `filterAddress` parameter each time. Note that you must execute the `dspCLListen` routine asynchronously to take advantage of this feature.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errState`	–1278	Not a connection listener
`errAborted`	–1279	Request aborted by the `dspRemove` routine
`errRefNum`	–1280	Bad connection reference number

dspCLDeny

The `dspCLDeny` routine denies a connection request from a remote connection end. You use the `PBControl` function to call the `dspCLDeny` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block.

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OsErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspCLDeny` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`remoteCID`	`Integer`	The ID of the remote connection end.
→	`remoteAddress`	`AddrBlock`	The remote internet address.

Field descriptions

`csCode`	The routine selector, always `dspCLDeny` for this routine.
`ccbRefNum`	The CCB reference number for the connection listener that received the request. This is the CBB number that the `dspCLInit` routine returned for the connection listener when you established a connection listener.
`remoteCID`	The ID of the remote connection end. The `dspCLListen` routine returns this value.
`remoteAddress`	The internet address of the remote connection end. The `dspCLListen` routine returns this value.

DESCRIPTION

A connection server uses the `dspCLDeny` routine to inform a remote connection end that its request to open a connection cannot be honored. If you want your connection listener to continue to listen for further connection requests, you must call the `dspCLListen` request again after you call `dspCLDeny`.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errState`	–1278	Not a connection listener
`errAborted`	–1279	Request aborted by the `dspRemove` routine
`errRefNum`	–1280	Bad connection reference number

dspCLRemove

The `dspCLRemove` routine closes a connection end used as a connection listener. You use the `PBControl` function to call the `dspCLRemove` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspCLRemove` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`abort`	`Byte`	A value specifying to abort outstanding requests if not 0.

Field descriptions

`csCode`	The routine selector, always `dspCLRemove` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspCLInit` routine returned.
`abort`	A value directing ADSP whether or not to cancel any outstanding listen and deny requests. If this value is nonzero, ADSP cancels outstanding `dspCLListen` and `dspCLDeny` requests. If this value is 0, ADSP does not cancel these requests.

DESCRIPTION

The `dspCLRemove` routine closes a connection end used as a connection listener. After you call the `dspCLRemove` routine, you can release the memory that you allocated for the CCB if you do not intend to reopen the connection end.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errRefNum`	–1280	Bad connection reference number

Maintaining an ADSP Connection and Using It to Exchange Data

Once you have established a connection end and opened a connection, you can send and receive data over the connection. You can use the routines in this section to

- determine the status of a connection
- read bytes from the connection end's receive queue
- write bytes to the connection end's send queue and transmit them to the remote connection end
- send an attention message to the remote connection end
- discard all data that has been sent but not yet delivered, and reset the connection

dspStatus

The `dspStatus` routine returns the number of bytes waiting to be read and sent and the amount of space available in the send and receive queues. You use the `PBControl` function to call the `dspStatus` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspStatus` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
←	`statusCCB`	`TPCCB`	A pointer to the CCB.
←	`sendQPending`	`Integer`	Bytes waiting to be sent or acknowledged.
←	`sendQFree`	`Integer`	Available send queue in bytes.
←	`recvQPending`	`Integer`	Bytes waiting to be read from queue.
←	`recvQFree`	`Integer`	Available receive queue in bytes.

Field descriptions

`csCode`	The routine selector, always `dspStatus` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspInit` routine returned.
`statusCCB`	A pointer to the CCB of the connection specified by the `ccbRefNum` parameter value.
`sendQPending`	The number of bytes of data that are in the send queue waiting to be sent, including 1 byte for each logical end-of-message (EOM) indicator in the send queue. (ADSP counts 1 byte for each EOM, even though no actual data corresponds to the EOM indicator.) The send queue contains all data that has been sent to ADSP for transmission and that has not yet been acknowledged. Some of the data in the send queue might have already been transmitted, but ADSP retains it in the send queue until the remote connection end acknowledges its receipt in case the data has to be retransmitted.

`sendQFree`	The number of bytes available in the send queue for additional data.
`recvQPending`	The number of bytes in the receive queue, including 1 byte for each EOM if the EOM bit is set in an ADSP packet header. The receive queue contains all of the data that has been received by the connection end but not yet read by the connection end's client.
`recvQFree`	The number of bytes available in the receive queue for additional data.

DESCRIPTION

The `dspStatus` routine provides information about an open connection. In addition to returning the number of bytes waiting to be read and sent and the space available in the send and receive queues, this routine also returns a pointer to the CCB, which contains information about the state of the connection end and about connection events received by the connection end. For more information about the CCB, see "Creating and Using a Connection Control Block" on page 5-12 and "The ADSP Connection Control Block Record" beginning on page 5-35.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errRefNum`	–1280	Bad connection reference number

dspRead

The `dspRead` routine reads data from a connection end's receive queue and writes the data to a buffer that you specify. You use the `PBControl` function to call the `dspRead` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspRead` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`reqCount`	`Integer`	The requested number of bytes.
←	`actCount`	`Integer`	The actual number of bytes read.
→	`dataPtr`	`Ptr`	A pointer to the data buffer.
←	`eom`	`Byte`	A flag indicating the end of message.

Field descriptions

`csCode`	The routine selector, always `dspRead` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspInit` routine returned.
`reqCount`	The number of bytes that ADSP is to read.
`actCount`	The actual number of bytes that ADSP read.
`dataPtr`	A pointer to the buffer into which ADSP is to place the data.
`eom`	A flag indicating if the last byte that ADSP read was a logical end-of-message indicator. If the last byte constitutes an EOM, ADSP sets this parameter to 1. If not, it sets this parameter to 0.

DESCRIPTION

The `dspRead` routine reads data from an ADSP connection. You can continue to read bytes as long as data is in the receive queue, even after you have called the `dspClose` routine or after the remote connection end has called the `dspClose` or `dspRemove` routine. The `dspRead` routine completes execution when it has read the number of bytes you specify or when it encounters an end of message (that is, the last byte of data in an ADSP packet that has the EOM bit set in the packet header).

You can call the `dspStatus` routine to determine the number of bytes remaining to be read from the read queue, or you can continue to call the `dspRead` routine until the `actCount` and `eom` parameters both return 0.

If either end closes the connection before you call the `dspRead` routine, the command reads whatever data is available and returns the actual amount of data read in the `actCount` parameter. If the connection is closed and there is no data in the receive queue, the `dspRead` routine returns the `noErr` result code with the `actCount` parameter set to 0 and the `eom` parameter set to 0.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errFwdReset`	–1275	Read terminated by forward reset
`errState`	–1278	State isn't open, closing, or closed
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number

dspWrite

The `dspWrite` routine writes bytes into a connection end's send queue for ADSP or ASDSP to transmit across a connection. When ASDSP is used and the encrypt bit is set, ASDSP encrypts the data before sending it. You use the `PBControl` function to call the `dspWrite` routine. See "Routines" on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspWrite` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`reqCount`	`Integer`	The requested number of bytes.
←	`actCount`	`Integer`	The actual number of bytes written.
→	`dataPtr`	`Ptr`	A pointer to the data buffer.
→	`eom`	`Byte`	For ADSP: a flag indicating end of message. For ASDSP: a flag indicating end of message or encryption.
→	`flush`	`Byte`	A flag indicating whether to send buffered data.

Field descriptions

`csCode`	The routine selector, always `dspWrite` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspInit` routine returned.
`reqCount`	The number of bytes to write.
`actCount`	The actual number of bytes written to the send queue.
`dataPtr`	A pointer to the buffer from which ADSP or ASDSP should read the data that is to be sent.
`eom`	For ADSP, a flag indicating if the last byte written to the send queue was a logical end-of-message indicator. If the last byte constitutes an EOM, you set this parameter to 1. If not, you set this parameter to 0. The high-order bits of the `eom` parameter are reserved for use by ADSP; you must leave these bits equal to 0. For ASDSP, if this is a secure connection, this field constitutes two single-bit flags instead of a zero/nonzero byte. If set to 1, bit 0 indicates the end of message; if set to 1, bit 1 turns on encryption. Note that ASDSP checks this flag on the first write of the connection and the first write following a write for which the end-of-message flag (bit 0 of the `eom` field) is set.
`flush`	A flag indicting whether or not ADSP or ASDSP should immediately send the data in the send queue to the remote connection. Set `flush` to 1 to cause ADSP or ASDSP to immediately transmit any data in the send queue that has not already been transmitted. Set `flush` to 0 to

allow data to accumulate in the send queue until another condition occurs that causes data to be transmitted. The high-order bits of the `flush` parameter are reserved for use by ADSP or ASDSP; you must leave these bits equal to 0.

DESCRIPTION

The `dspWrite` routine sends data across an ADSP or ASDSP connection. The send queue contains all data that has been sent to ADSP or ASDSP for transmission and that has not yet been acknowledged. Some of the data in the send queue might have already been transmitted, but ADSP or ASDSP retains it in the send queue until the remote connection end acknowledges its receipt in case the data has to be retransmitted. The `dspWrite` routine completes execution when it has copied all of the data from the data buffer into the send queue.

ADSP or ASDSP transmits the data in the send queue when the remote connection end has room to accept the data and 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. (You use the `sendBlocking` parameter to the `dspOptions` routine to set the blocking factor.)
- The send timer expires.
- A connection event requires that the local connection end send an acknowledgment packet to the remote connection end.

For an ADSP `dspWrite` call, you can set the `reqCount` parameter to 0 and the `eom` parameter to 1 to indicate that the last byte you sent the previous time you called the `dspWrite` routine was the end of the message. You can set the `reqCount` parameter to a value larger than the size of the send queue. If you do so, the `dspWrite` routine writes as much data as it can into the send queue, sends the data and waits for acknowledgment, and then writes more data into the send queue until it has written the amount of data you requested. In this case, the routine does not complete execution until it has finished writing all of the data into the send queue.

For an ASDSP `dspWrite` call, you can set the encrypt bit of the `eom` field (bit 1) of the DSP parameter block. Note that ASDSP checks this flag on the first write of the connection or the first write following a write for which the end-of-message flag (bit 0 of the `eom` field) is set. You can set the end-of-message bit (bit 0) of the `eom` field to indicate the end of the message.

- To set the encrypt bit, you use the `dspEncryptMask` mask or the `dspEncryptBit` constant.
- To set the end-of-message bit, you use the `dspEOMMask` mask or the `dspEOMBit` constant.

Set the `flush` parameter to 1 to cause ADSP to immediately transmit any data in the send queue that has not already been transmitted. Set the `flush` parameter to 0 to allow data to accumulate in the send queue until another condition occurs that causes data to be transmitted.

If you want to encrypt all messages, you can simply set the encrypt bit on all calls to the `dspWrite` function.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errState`	–1278	Connection is not open
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number

dspAttention

The `dspAttention` routine sends an attention code and an attention message to the remote connection end. You use the `PBControl` function to call the `dspAttention` routine. See “Routines” on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspAttention` for this function.
→	`ccbRefNum`	`Integer`	The CCB reference number.
→	`attnCode`	`Integer`	The client attention code.
→	`attnSize`	`Integer`	The size of attention data in bytes.
→	`attnData`	`Ptr`	A pointer to attention data.

Field descriptions

`csCode`	The routine selector, always `dspAttention` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspInit` routine returned.
`attnCode`	The 2-byte attention code that you wish to send to the remote connection end. You can use any value from $0000 through $EFFF for the attention code. The values $F000 through $FFFF are reserved for use by ADSP.
`attnSize`	The size in bytes of the attention message you wish to send.
`attnData`	A pointer to the attention message. The attention message can be any size from 0 through 570 bytes. There are no restrictions on the content of the attention message.

DESCRIPTION

The `dspAttention` routine sends an attention code and message. Attention codes and attention messages can have any meaning that your application and the application at the remote connection end both recognize. The purpose of attention codes and messages is to allow clients of ADSP to send messages outside the normal data stream.

For example, if a connection end on a mainframe computer is connected to several connection ends in Macintosh computers being used as remote terminals, the mainframe computer might wish to inform the remote terminals that all connections will be terminated in ten minutes. The mainframe application could send an attention message to each of the remote terminals informing them of this fact, and the terminal emulation programs in the Macintosh computers could then display an alert message on the screen so that the users could prepare to shut down.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errAttention`	–1276	Attention message too long
`errState`	–1278	Connection is not open
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number

dspReset

The `dspReset` routine clears all the data associated with the connection that the remote connection client has not already read and resynchronizes the connection. You use the `PBControl` function to call the `dspReset` routine. See “Routines” on page 5-43 for a description of the `PBControl` function.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`ioCRefNum`	`Integer`	The driver reference number.
→	`csCode`	`Integer`	Always `dspReset` for this routine.
→	`ccbRefNum`	`Integer`	The CCB reference number.

Field descriptions

`csCode`	The routine selector, always `dspReset` for this routine.
`ccbRefNum`	The connection control block (CCB) reference number that the `dspInit` routine returned.

DESCRIPTION

The `dspReset` routine causes ADSP to discard all data in the send queue, all data in transit to the remote connection end, and all data in the remote connection end's receive queue that the client has not yet read. This process is known as a *forward reset*. ADSP then resynchronizes the connection. You can determine that your connection end has received a forward reset and has discarded all data in the receive queue by checking the `eFwdReset` flag in the `userFlags` field of the CCB. For information on the CCB, see "Connections, Connection Ends, and Connection States" beginning on page 5-6.

ASSEMBLY-LANGUAGE INFORMATION

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

RESULT CODES

`noErr`	0	No error
`errState`	–1278	Connection is not open
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number

Summary of ADSP

Pascal Summary

Constants

```
CONST
   {ADSP routine selectors}
   dspInit                  = 255;        {create a new connection end}
   dspRemove                = 254;        {remove a connection end}
   dspOpen                  = 253;        {open a connection}
   dspClose                 = 252;        {close a connection}
   dspCLInit                = 251;        {create a connection listener}
   dspCLRemove              = 250;        {remove a connection listener}
   dspCLListen              = 249;        {post a listener request}
   dspCLDeny                = 248;        {deny an open-connection request}
   dspStatus                = 247;        {get status of connection end}
   dspRead                  = 246;        {read data from the connection}
   dspWrite                 = 245;        {write data on the connection}
   dspAttention             = 244;        {send an attention message}
   dspOptions               = 243;        {set connection end options}
   dspReset                 = 242;        {forward reset the connection}
   dspNewCID                = 241;        {generate a CID for a }
                                          { connection end}
   sdspOpen                 = 229;        {open a secure connection}

   {ADSP connection-opening modes}
   ocRequest                = 1;          {request a connection with a }
                                          { remote connection end}
   ocPassive                = 2;          {wait for a connection request }
                                          { from remote connection end}
   ocAccept                 = 3;          {accept request as delivered by }
                                          { listener}
   ocEstablish              = 4;          {consider connection to be open}

   {ADSP connection end states}
   sListening               = 1;          {for connection listeners}
   sPassive                 = 2;          {waiting for a connection }
                                          { request from remote }
                                          { connection end}
   sOpening               = 3;          {requesting a connection }
                                        { with remote connection end}
   sOpen                  = 4;          {connection is open}
   sClosing               = 5;          {connection is being torn down}
   sClosed                = 6;          {connection end state is closed}

   {ASDSP end-of-message and encrypt flags and masks}
   dspEncryptBit          = 1;          {set to encrypt message}
   dspEOMBit              = 0;          {set if EOM at end of write}
   dspEOMMask             = $1;         {mask for setting the EOM bit}
   dspEncryptMask         = $2;         {mask for setting the encrypt bit}

   {ADSP client event flags}
   eClosed                = $80;        {received connection-closed event}
   eTearDown              = $40;        {closed due to broken connection}
   eAttention             = $20;        {received attention message}
   eFwdReset              = $10;        {received forward reset event}

   {miscellaneous ADSP constants}
   attnBufSize            = 570;        {size of client attention buffer}
   minDSPQueueSize        = 100;        {minimum size of receive or }
                                        { send queue}

   {driver control ioResults}
   errRefNum              = -1280;      {bad connection refNum}
   errAborted             = -1279;      {control call was aborted}
   errState               = -1278;      {bad connection state for this }
                                        { operation}
   errOpening             = -1277;      {open connection request failed}
   errAttention           = -1276;      {attention message too long}
   errFwdReset            = -1275;      {read terminated by forward reset}
   errDSPQueueSize        = -1274;      {DSP read/write queue too small}
   errOpenDenied          = -1273;      {open connection request denied}
```

Data Types

The ADSP Connection Control Block Record

```
TYPE TRCCB =
   PACKED RECORD
      ccbLink:          TPCCB;      {link to next CCB}
      refNum:           Integer;    {reference number}
      state:            Integer;    {state of the connection end}
      userFlags:         Byte;       {user flags for connection}
      localSocket:       Byte;       {local socket number}
      remoteAddress:     AddrBlock;  {remote end internet address}
      attnCode:          Integer;    {attention code received}
      attnSize:          Integer;    {size of attention data}
      attnPtr:           Ptr;        {pointer to attention data}
      reserved:          PACKED ARRAY[1..220] OF Byte;
                                     {reserved for use by ADSP}
   END;
```

The Address Block Record

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

The DSP Parameter Block

```
TYPE DSPParamBlock =
   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}
      ioCRefNum:         Integer;    {driver reference number}
      csCode:            Integer;    {primary command code}
      qStatus:           LongInt;    {reserved}
      ccbRefNum:         Integer;    {CCB reference number}
   CASE Integer OF
   dspInit, dspCLInit:
      (ccbPtr:           TPCCB;      {pointer to CCB}
      userRoutine:       ProcPtr;    {pointer to user routine}
      sendQSize:         Integer;    {size of send queue}
      sendQueue:         Ptr;        {pointer to send queue}
      recvQSize:         Integer;    {size of receive queue}
      recvQueue:         Ptr;        {pointer to receive queue}
      attnPtr:           Ptr;        {pointer to attention-message buffer}
      localSocket:      Byte;       {local socket number}
      filler1:          Byte);      {filler for proper alignment}
   dspOpen, dspCLListen, dspCLDeny:
      (localCID:        Integer;    {local connection ID}
      remoteCID:        Integer;    {remote connection ID}
      remoteAddress:    AddrBlock;  {remote internet address}
      filterAddress:    AddrBlock;  {address filter}
      sendSeq:          LongInt;    {send sequence number}
      sendWindow:       Integer;    {size of remote buffer}
      recvSeq:          LongInt;    {receive sequence number}
      attnSendSeq:      LongInt;    {attention send seq number}
      attnRecvSeq:      LongInt;    {attention receive seq num}
      ocMode:           Byte;       {connection-opening mode}
      ocInterval:       Byte;       {interval bet. open requests}
      ocMaximum:        Byte;       {retries of open-conn req}
      filler2:          Byte);      {filler for proper alignment}
   dspClose, dspRemove:
      (abort:           Byte;       {abort send requests}
      filler3:          Byte);      {filler for proper alignment}
   dspStatus:
      (statusCCB:       TPCCB;      {pointer to CCB}
      sendQPending:     Integer;    {bytes waiting in send queue}
      sendQFree:        Integer;    {available send-queue buffer}
      recvQPending:     Integer;    {bytes in receive queue}
      recvQFree:        Integer);   {avail receive-queue buffer}
   dspRead, dspWrite:
      (reqCount:        Integer;    {requested number of bytes}
      actCount:         Integer;    {actual number of bytes}
      dataPtr:          Ptr;        {pointer to data buffer}
      eom:              Byte;       {1 if end of message}
      flush:            Byte);      {1 to send data now}
   dspAttention:
      (attnCode:        Integer;    {client attention code}
      attnSize:         Integer;    {size of attention data}
      attnData:         Ptr;        {pointer to attention data}
      attnInterval:     Byte;       {reserved}
      filler4:          Byte);      {filler for proper alignment}
   dspOptions:
      (sendBlocking:    Integer;    {send-blocking threshold}
      sendTimer:        Byte;       {reserved}
      rtmtTimer:        Byte;       {reserved}
      badSeqMax:        Byte;       {retransmit advice threshold}
      useCheckSum:      Byte);      {DDP checksum for packets}
   dspNewCID:
      (newCID:          Integer);   {new connection ID}
   END;

DSPPBPtr = ^DSPParamBlock;
```

The ASDSP Parameter Block

```
TYPE SDSPParamBlock =
   PACKED RECORD
   CASE INTEGER OF
      1: (dspParamBlock: DSPParamBlock);
      2: (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}
         ioCRefNum:     Integer;       {adsp driver refNum}
         csCode:        Integer;       {asdsp driver control code}
         qStatus:       Longint;       {reserved}
         ccbRefNum:     Integer;       {connection end refNum}
         secureParams:  TRSecureParams);
                                       {parameters for dspOpenSecure}
   END;

SDSPPBPtr = ^SDSPParamBlock;
```

The TRSecureParams Record

```
TYPE  TRSecureParams =
   PACKED RECORD
      localCID:         Integer;       {local connection ID}
      remoteCID:        Integer;       {remote connection ID}
      remoteAddress:    AddrBlock;     {address of remote end}
      filterAddress:    AddrBlock;     {address filter}
      sendSeq:          Longint;       {local send sequence number}
      sendWindow:       Integer;       {send window size}
      recvSeq:          Longint;       {receive sequence number}
      attnSendSeq:      Longint;       {attention send sequence number}
      attnRecvSeq:      Longint;       {attention receive sequence number}
      ocMode:           Byte;          {open connection mode}
      ocInterval:       Byte;          {open connection request }
                                       { retry interval}
      ocMaximum:        Byte;          {open connection request }
                                       { retry maximum}
      secure:           Boolean;       {for initiator, TRUE if session is }
                                       { authenticated}
                                       {for recipient, TRUE if session was }
                                       { authenticated}
      sessionKey:       AuthKeyPtr;    {encryption key for session}
      credentialsSize:  Longint;       {length of credentials}
      credentials:      Ptr;           {pointer to credentials}
      workspace:        Ptr;           {pointer to workspace for }
                                       { connection. Align on }
                                       { even boundary and }
                                       { length = sdspWorkSize}
      recipient:        AuthIdentity;  {identity of recipient }
                                       { or initiator if active mode}
      issueTime:        UTCTime;       {time when credentials were issued}
      expiry:           UTCTime;       {time when credentials expire}
      initiator:        RecordIDPtr;   {RecordID of initiator returned in }
                                       { the buffer pointed to by this field}
      hasIntermediary:  Boolean;       {set if credentials has an }
                                       { intermediary}
      intermediary:     RecordIDPtr;   {Record ID of intermediary returned}
   END;
```

C Summary

Constants

```
/*workspace used internally by ASDSP for the sdspOpen call*/
#define sdspWorkSize          2048        /*size of ASDSP workspace*/

enum{                                     /*ADSP routine selectors*/
   dspInit               = 255,           /*create a new connection end*/
   dspRemove             = 254,           /*remove a connection end*/
   dspOpen               = 253,           /*open a connection*/
   dspClose              = 252,           /*close a connection*/
   dspCLInit             = 251,           /*create a connection listener*/
   dspCLRemove           = 250,           /*remove a connection listener*/
   dspCLListen           = 249,           /*post a listener request*/
   dspCLDeny              = 248,        /*an open-connection request*/
   dspStatus              = 247,        /*get status of connection end*/
   dspRead                = 246,        /*read data from the connection*/
   dspWrite               = 245,        /*write data on the connection*/
   dspAttention           = 244,        /*send an attention message*/
   dspOptions             = 243,        /*set connection end options*/
   dspReset               = 242,        /*forward reset the connection*/
   dspNewCID              = 241,        /*generate a CID for a */
                                        /* connection end*/
   sdspOpen               = 229;        /*open a secure connection*/

enum {                                  /*ADSP connection-opening modes*/
   ocRequest              = 1,          /*request a connection with a */
                                        /* remote connection end*/
   ocPassive              = 2,          /*wait for a connection request */
                                        /* from remote connection end*/
   ocAccept               = 3,          /*accept request as delivered by */
                                        /* listener*/
   ocEstablish            = 4};         /*consider connection to be */
                                        /* open*/

enum {                                  /*ADSP connection end states*/
   sListening             = 1,          /*for connection listeners*/
   sPassive               = 2,          /*waiting for a connection */
                                        /* request from remote */
                                        /* connection end*/
   sOpening               = 3,          /*requesting a connection */
                                        /* with remote connection end*/
   sOpen                  = 4,          /*connection is open*/
   sClosing               = 5,          /*connection is being torn down*/
   sClosed                = 6};         /*connection end state */
                                        /* is closed*/

/*ASDSP end-of-message and encrypt flags and masks*/
enum {
   dspEOMBit              = 0,          /*set if EOM at end of write*/
   dspEncryptBit          = 1};         /*set to encrypt message*/

enum {
   dspEOMMask             = 1<<dspEOMBit,
   dspEncryptMask         = 1<<dspEncryptBit
};
```

```
enum {                                  /*ADSP client event flags*/
   eClosed                = $80,        /*received connection-closed */
                                        /* event*/
   eTearDown              = $40,        /*closed due to broken */
                                        /* connection*/
   eAttention             = $20,        /*received attention message*/
   eFwdReset              = $10};       /*received forward reset event*/

enum {                                  /*miscellaneous ADSP constants*/
   attnBufSize            = 570,        /*size of client attention */
                                        /* buffer*/
   minDSPQueueSize        = 100};       /*minimum size of receive or */
                                        /* send queue*/

enum {                                  /*driver control ioResults*/
   errRefNum              = -1280,      /*bad connection refNum*/
   errAborted             = -1279,      /*control call was aborted*/
   errState               = -1278,      /*bad connection state for this */
                                        /* operation*/
   errOpening             = -1277,      /*open connection request */
                                        /* failed*/
   errAttention           = -1276,      /*attention message too long*/
   errFwdReset            = -1275,      /*read terminated */
                                        /* by forward reset*/
   errDSPQueueSize        = -1274,      /*DSP read/write queue */
                                        /* too small*/
   errOpenDenied          = -1273};     /*open connection request */
                                        /* denied*/
```

Data Types

The ADSP Connection Control Block Record

```
struct TRCCB {
   unsigned char     *ccbLink;         /*link to next CCB*/
   unsigned short    refNum;           /*reference number*/
   unsigned short    state;            /*state of the connection end*/
   unsigned char     userFlags;        /*user flags for connection*/
   unsigned char     localSocket;      /*local socket number*/
   AddrBlock         remoteAddress;    /*remote end internet address*/
   unsigned short    attnCode;         /*attention code received*/
   unsigned short    attnSize;         /*size of attention data*/
   unsigned char     *attnPtr;          /*pointer to attention data*/
   unsigned char     reserved[220];     /*reserved*/
};

typedef struct TRCCB TRCCB;
typedef TRCCB *TPCCB;
```

The Address Block Record

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

typedef struct AddrBlock AddrBlock;
```

Parameter Block for dspInit and dspCLInit

```
struct TRinitParams {
   TPCCB             ccbPtr;         /*pointer to connection control block*/
   ProcPtr           userRoutine;    /*client routine to call on event*/
   unsigned short    sendQSize;      /*size of send queue (0..64K bytes)*/
   unsigned char     *sendQueue;     /*client passed send queue buffer*/
   unsigned short    recvQSize;      /*size of receive queue */
                                     /* (0..64K bytes)*/
   unsigned char     *recvQueue;     /*client passed receive queue buffer*/
   unsigned char     *attnPtr;       /*client passed receive attention */
                                     /* buffer*/
   unsigned char     localSocket;    /*local socket number*/
};

typedef struct TRinitParams TRinitParams;
```

Parameter Block for dspOpen, dspCLListen, and dspCLDeny

```
struct TRopenParams {
   unsigned short    localCID;          /*local connection ID*/
   unsigned short    remoteCID;         /*remote connection ID*/
   AddrBlock         remoteAddress;     /*address of remote end*/
   AddrBlock         filterAddress;     /*address filter*/
   unsigned long     sendSeq;           /*local send sequence number*/
   unsigned short    sendWindow;        /*send window size*/
   unsigned long     recvSeq;           /*receive sequence number*/
   unsigned long     attnSendSeq;      /*attention send sequence number*/
   unsigned long     attnRecvSeq;      /*attention receive sequence */
                                       /* number*/
   unsigned char     ocMode;           /*open connection mode*/
   unsigned char     ocInterval;       /*open connection request retry */
                                       /* interval*/
   unsigned char     ocMaximum;        /*open connection request retry */
};                                     /* maximum*/

typedef struct TRopenParams TRopenParams;
```

Parameter Block for dspClose and dspRemove

```
struct TRcloseParams {
   unsigned char        abort; /*abort connection immediately if nonzero*/
};

typedef struct TRcloseParams TRcloseParams;
```

Parameter Block for dspStatus

```
struct TRstatusParams {
   TPCCB                ccbPtr;        /*pointer to ccb*/
   unsigned short       sendQPending;  /*pending bytes in send queue*/
   unsigned short       sendQFree;     /*available buffer space in send */
                                       /* queue*/
   unsigned short       recvQPending;  /*pending bytes in receive queue*/
   unsigned short       recvQFree;     /*available buffer space in */
};                                     /* receive queue*/

typedef struct TRstatusParams TRstatusParams;
```

Parameter Block for dspRead and dspWrite

```
struct TRioParams {
   unsigned short       reqCount;      /*requested number of bytes*/
   unsigned short       actCount;      /*actual number of bytes*/
   unsigned char        *dataPtr;      /*pointer to data buffer*/
   unsigned char        eom;           /*indicates logical end of message*/
   unsigned char        flush;         /*send data now*/
};

typedef struct TRioParams TRioParams;
```

Parameter Block for dspAttention

```
struct TRattnParams {
   unsigned short      attnCode;      /*client attention code*/
   unsigned short      attnSize;      /*size of attention data*/
   unsigned char       *attnData;     /*pointer to attention data*/
   unsigned char       attnInterval;  /*retransmit timer in 10-tick */
                                      /* intervals*/
};

typedef struct TRattnParams TRattnParams;
```

Parameter Block for dspOptions

```
struct TRoptionParams {
   unsigned short      sendBlocking;  /*quantum for data packets*/
   unsigned char       sendTimer;     /*send timer in 10-tick intervals*/
   unsigned char       rtmtTimer;     /*retransmit timer in 10-tick */
                                      /* intervals*/
   unsigned char       badSeqMax;     /*threshold for sending retransmit */
                                      /* advice*/
   unsigned char       useCheckSum;   /*use ddp packet checksum*/
};

typedef struct TRoptionParams TRoptionParams;
```

Parameter Block for dspNewCID

```
struct TRnewcidParams {
   unsigned short      newcid;        /*new connection ID returned*/
};

typedef struct TRnewcidParams TRnewcidParams;
```

The DSP Parameter Block

```
struct DSPParamBlock {
   struct QElem     *qLink;           /*reserved*/
   short            qType;            /*reserved*/
   short            ioTrap;           /*reserved*/
   Ptr              ioCmdAddr;        /*reserved*/
   ProcPtr          ioCompletion;     /*pointer to completion routine*/
   OSErr            ioResult;         /*routine result*/
   char             *ioNamePtr;       /*reserved*/
   short            ioVRefNum;        /*reserved*/
   short            ioCRefNum;        /*ADSP driver refNum*/
   short            csCode;           /*ADSP driver control code*/
   long             qStatus;          /*reserved*/
   short            ccbRefNum;
union{
   TRinitParams     initParams;       /*dspInit, dspCLInit*/
   TRopenParams     openParams;       /*dspOpen, dspCLListen, dspCLDeny*/
   TRcloseParams    closeParams;      /*dspClose, dspRemove*/
   TRioParams       ioParams;         /*dspRead, dspWrite*/
   TRattnParams     attnParams;       /*dspAttention*/
   TRstatusParams   statusParams;     /*dspStatus*/
   TRoptionParams   optionParams;     /*dspOptions*/
   TRnewcidParams   newCIDParams;     /*dspNewCID*/
   } u;
};

typedef struct DSPParamBlock DSPParamBlock;
typedef DSPParamBlock *DSPPBPtr;
```

The ASDSP Parameter Block

```
struct TRSecureParams {
   unsigned short   localCID;         /*local connection ID*/
   unsigned short   remoteCID;        /*remote connection ID*/
   AddrBlock        remoteAddress;    /*address of remote end*/
   AddrBlock        filterAddress;    /*address filter*/
   unsigned long    sendSeq;          /*local send sequence number*/
   unsigned short   sendWindow;       /*send window size*/
   unsigned long    recvSeq;          /*receive sequence number*/
   unsigned long    attnSendSeq;      /*attention send sequence number*/
   unsigned long    attnRecvSeq;      /*attention receive sequence */
                                      /* number*/
   unsigned char    ocMode;           /*open connection mode*/
   unsigned char    ocInterval;       /*open connection request retry */
                                      /* interval*/
   unsigned char    ocMaximum;        /*open connection request retry */
                                      /* maximum*/
   Boolean          secure;           /*TRUE if session was */
                                      /* authenticated*/
   AuthKeyPtr       sessionKey;       /*encryption key for session*/
   unsigned         longcredentialsSize;
                                      /*length of credentials*/
   Ptr              credentials;      /*pointer to credentials*/
   Ptr               workspace;        /*pointer to workspace for */
                                       /* connection. align on even */
                                       /* boundary and length equals */
                                       /* sdspWorkSize*/
   AuthIdentity      recipient;        /*identity of recipient */
                                       /* (or initiator if active mode)*/
   UTCTime           issueTime;        /*when credentials were issued*/
   UTCTime           expiry;           /*when credentials expire*/
   RecordIDPtr       initiator;        /*pointer to RecordID of */
                                       /* initiator returned*/
   Boolean           hasIntermediary;  /*is set if credentials */
                                       /* have an intermediary*/
   RecordIDPtr       intermediary;     /*pointer to RecordID of */
                                       /* intermediary returned*/
   };
```

The TRSecureParams Record

```
typedef struct TRSecureParams TRSecureParams;

struct SDSPParamBlock {
      struct QElem      *qLink;        /*reserved*/
      short             qType;         /*reserved*/
      short             ioTrap;        /*reserved*/
      Ptr               ioCmdAddr;     /*reserved*/
      ProcPtr           ioCompletion;
                                       /*pointer to completion routine*/
      OSErr             ioResult;      /*routine result*/
      char              *ioNamePtr;    /*reserved*/
      short             ioVRefNum;     /*reserved*/
      short             ioCRefNum;     /*ADSP driver refNum*/
      short             csCode;        /*ADSP driver control code*/
      long              qStatus;       /*ADSP internal use*/
      short             ccbRefNum;     /*connection end refNum*/

   union {
      TRinitParams      initParams;    /*dspInit, dspCLInit*/
      TRopenParams      openParams;    /*dspOpen, dspCLListen, dspCLDeny*/
      TRcloseParams     closeParams;   /*dspClose, dspRemove*/
      TRioParams        ioParams;      /*dspRead, dspWrite*/
      TRattnParams      attnParams;    /*dspAttention*/
      TRstatusParams    statusParams;  /*dspStatus*/
      TRoptionParams    optionParams;  /*dspOptions*/
      TRnewcidParams    newCIDParams;  /*dspNewCID*/
      TRSecureParams    secureParams;  /*dspOpenSecure*/
   } u;
};

typedef struct SDSPParamBlock SDSPParamBlock;
typedef SDSPParamBlock *SDSPPBPtr;
```

Assembly-Language Summary

Constants

ADSP Queue Element Equates and Sizes

```
csQStatus      EQU      CSParam        ;ADSP internal use
csCCBRef       EQU      csQStatus+4    ;refnum of ccb
```

Command Codes

```
dspInit        EQU      255            ;create a new connection end
dspRemove      EQU      254            ;remove a connection end
dspOpen        EQU      253            ;open a connection
dspClose       EQU      252            ;close a connection
dspCLInit      EQU      251            ;create a connection listener
dspCLRemove    EQU      250            ;remove a connection listener
dspCLListen    EQU      249            ;post a listener request
dspCLDeny      EQU      248            ;deny an open connection request
dspStatus      EQU      247            ;get status of connection end
dspRead        EQU      246            ;read data from the connection
dspWrite       EQU      245            ;write data on the connection
dspAttention   EQU      244            ;send an attention message
dspOptions     EQU      243            ;set connection end options
dspReset       EQU      242            ;forward reset the connection
dspNewCID      EQU      241            ;generate a cid for a connection end
sdspOpen       EQU      229            ;open a secure connection
```

Open Connection Modes

```
ocRequest      EQU     1              ;request a connection with remote
ocPassive      EQU     2              ;wait for a connection request from
                                      ; remote
ocAccept       EQU     3              ;accept request as delivered by
                                      ; listener
ocEstablish    EQU     4              ;consider connection to be open
```

Connection States

```
sListening     EQU     1              ;for connection listeners
sPassive       EQU     2              ;waiting for a connection request
                                      ; from remote
sOpening       EQU     3              ;requesting a connection with remote
sOpen          EQU     4              ;connection is open
sClosing       EQU     5              ;connection is being torn down
sClosed        EQU     6              ;connection end state is closed
```

Client Event Flags (Bit-Mask)

```
eClosed        EQU     $80            ;received connection closed advice
eTearDown      EQU     $40            ;closed due to broken connection
eAttention     EQU     $20            ;received attention message
eFwdReset      EQU     $10            ;received forward reset advice
```

Miscellaneous Equates

```
attnBufSize    EQU     570            ;size of client attention message
minDSPQueueSize
               EQU     100            ;minimum size for both receive and
                                      ; send queues
sdspWorkSize   EQU     2048           ;size of ASDSP workspace
```

ASDSP Encrypt and End-of-Message Flags and Masks

```
dspEOMBit      EQU     0              ;set if EOM at end of write
dspEncryptBit  EQU     1              ;set to encrypt message
dspEncryptMask EQU     $1             ;mask for setting the encrypt bit
dspEOMMask     EQU     $2             ;mask for setting the EOM bit
```

Data Structures

ADSP Connection Control Block Data Structure

0	`ccbLink`	long	link to next CCB
4	`refNum`	word	reference number
6	`state`	word	state of the connection end
8	`userFlags`	byte	user flags for connection
9	`localSocket`	byte	local socket number
10	`remoteAddress`	long	internet address of remote end
14	`attnCode`	word	attention code received
16	`attnSize`	word	size of received attention data
18	`attnPtr`	long	pointer to received attention data
22	`reserved`	220 bytes	reserved

DPS Parameter Block Common Fields for ADSP and ASDSP

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	`ioCRefNum`	word	driver reference number
28	`qStatus`	long	reserved
32	`ccbRefNum`	word	reference number of CCB

dspInit and dspCLInit Parameter Variant

26	`csCode`	word	`dspInit` or `dspCLInit`
34	`ccbPtr`	long	pointer to CCB
38	`userRoutine`	long	pointer to routine to call on connection events
42	`sendQSize`	word	size in bytes of the send queue
44	`sendQueue`	long	pointer to send queue
48	`recvQSize`	word	size in bytes of the receive queue
50	`recvQueue`	long	pointer to receive queue
54	`attnPtr`	long	pointer to buffer for incoming attention messages
58	`localSocket`	byte	DDP socket number for this connection end

dspOptions Parameter Variant

16	`ioResult`	word	result code
24	`ioCRefNum`	word	driver reference number
26	`csCode`	word	always `dspOptions`
34	`sendBlocking`	word	send-blocking threshold
38	`badSeqMax`	byte	threshold to send retransmit advice
39	`useCheckSum`	byte	DDP checksum flag

dspOpen, dspCLListen, and dspCLDeny Parameter Variant

26	`csCode`	word	`dspOpen`, `dspCLListen`, or `dspCLDeny`
34	`localCID`	word	ID of this connection end
36	`remoteCID`	word	ID of remote connection end
38	`remoteAddress`	long	remote internet address
42	`filterAddress`	long	filter for open-connection requests
46	`sendSeq`	long	initial send sequence number
50	`sendWindow`	word	initial size of remote receive queue
52	`recvSeq`	long	initial receive sequence number
56	`attnSendSeq`	long	attention send sequence number
60	`attnRecvSeq`	long	attention receive sequence number
64	`ocMode`	byte	connection-opening mode
65	`ocInterval`	byte	interval between open requests
66	`ocMaximum`	byte	retries of open-connection request

sdspOpen Parameter Variant

26	`csCode`	word	`sdspOpen`
34	`localCID`	word	ID of this connection end
36	`remoteCID`	word	ID of remote connection end
38	`remoteAddress`	long	remote internet address
42	`filterAddress`	long	filter for open-connection requests
46	`sendSeq`	long	initial send sequence number
50	`sendWindow`	word	initial size of remote receive queue
52	`recvSeq`	long	not used for ASDSP
56	`attnSendSeq`	long	attention send sequence number
60	`attnRecvSeq`	long	not used for ASDSP
64	`ocMode`	byte	connection-opening mode
65	`ocInterval`	byte	interval between open requests
66	`ocMaximum`	byte	retries of open-connection request
68	`secure`	word	flag that determines if ASDSP authenticates the connection
70	`sessionKey`	long	pointer to the encryption key for the session
74	`credentialsSize`	long	length of credentials
78	`credentials`	long	pointer to credentials
82	`workspace`	long	pointer to workspace for connection
86	`recipient`	long	identity of recipient
90	`issueTime`	long	time when credentials were issued
94	`expiry`	long	time when credentials expire
98	`initiator`	long	pointer to record ID of initiator
102	`hasIntermediary`	word	`TRUE` if credentials have an intermediary
104	`intermediary`	long	pointer to record ID of intermediary

dspNewCID Parameter Variant

26	`csCode`	word	always `dspNewCID`
34	`newCID`	word	ID of new connection

dspClose, dspRemove, and dspCLRemove Parameter Variant

26	`csCode`	word	`dspClose`, `dspRemove`, or `dspCLRemove`
34	`abort`	byte	abort send requests or connection listener if not 0

dspStatus Parameter Variant

26	`csCode`	word	always `dspStatus`
34	`statusCCB`	pointer	pointer to CCB
38	`sendQPending`	word	bytes waiting to be sent or acknowledged
40	`sendQFree`	word	available send queue in bytes
42	`recvQPending`	word	bytes waiting to be read from queue
44	`recvQFree`	word	available receive queue in bytes

dspRead and dspWrite Parameter Variant

26	`csCode`	word	`dspRead` or `dspWrite`
34	`reqCount`	word	requested number of bytes
36	`actCount`	word	actual number of bytes read or written
38	`dataPtr`	pointer	pointer to data buffer
42	`eom`	byte	for ADSP: 1 if end of message; 0 otherwise for ASDSP: bit 0 = end of message; bit 1 turns on encryption, if set
43	`flush`	byte	1 to send data now; 0 otherwise

dspAttention and dspReset Parameter Variant

26	`csCode`	word	`dspAttention` or `dspReset`
34	`attnCode`	word	client attention code
36	`attnSize`	word	size of attention data in bytes
38	`attnData`	pointer	pointer to attention data

Result Codes

`noErr`	0	No error or unrecognized event code
`ddpSktErr`	–91	Error opening socket
`errOpenDenied`	–1273	Open request denied by recipient
`errDSPQueueSize`	–1274	Send or receive queue is too small
`errFwdReset`	–1275	Read terminated by forward reset
`errAttention`	–1276	Attention message too long
`errOpening`	–1277	Attempt to open connection failed
`errState`	–1278	Bad connection state for this operation
`errAborted`	–1279	Request aborted by `dspRemove` or `dspClose` routine
`errRefNum`	–1280	Bad connection reference number
`kOCEUnsupportedCredentialsVersion`	–1543	Credentials version not supported
`kOCEBadEncryptionMethod`	–1559	During the authentication process, the ASDSP implementations could not agree on an encryption method to be used (ASDSP can support multiple stream encryption methods. In Release 1, only RC4 and "no encryption" are supported.)
`kOCENoASDSPWorkSpace`	–1570	You passed `NIL` for the workspace parameter
`kOCEAuthenticationTrouble`	–1571	Authentication process failed