This chapter describes the AppleTalk Transaction Protocol (ATP) that you use to send a request from one application or process to another that can satisfy the request and respond to it. Because ATP is transaction-based—that is, the response data is bound to the request data and the exchange of information is limited to the transaction—you do not incur the overhead entailed in establishing, maintaining, and breaking a connection that is associated with connection-oriented protocols, such as ADSP. However, you can transfer only a limited amount of data using ATP.

You should read this chapter if you want to write an application that requires reliable delivery of data while allowing one side of the communication to ask the other side to perform a service and return a small amount of data.

For an overview of ATP 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 complete explanation of the ATP specification, see *Inside AppleTalk*, second edition.

About ATP

The AppleTalk Transaction Protocol offers a simple, efficient means of transferring *small* amounts of data across a network; it lets one network entity request information of another entity that possesses only the ability to respond to the request. ATP ensures that data is delivered without error or packet loss.

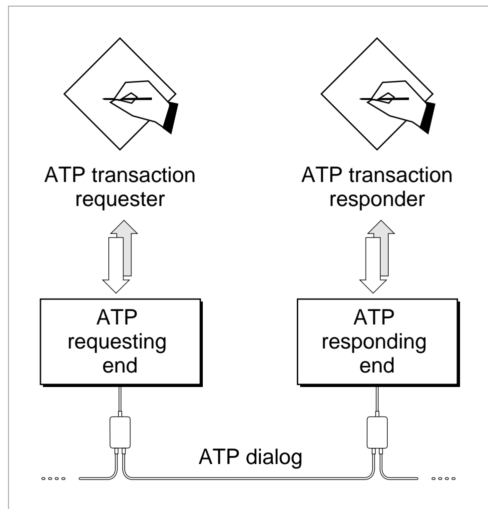
ATP communication is based on the concept of a ***transaction:*** one party, the ***requester,*** makes a request of another party, the ***responder,*** to perform a service and return a response. This discussion uses the term *requester* to refer to an application that uses ATP to make a request and *responder* to refer to an application that uses ATP to respond to a request.

When it receives a request, the responder application performs the necessary processing to service it and sends a response message back to the requester, completing the transaction. The response message can be data that reports the result of the transaction or information produced as a result of the processing. Here is how a basic transaction occurs:

- The requester application calls the .ATP interface, and the .ATP driver on the requester side sends the request to the .ATP driver on the responder side.
- The .ATP driver on the responder side passes the request to the responder application, which is listening for incoming .ATP requests.
- The responder application satisfies the request and prepares a response, then calls the ATP interface to transmit the response via the .ATP driver back to the requester application.

Figure 6-1 shows this interaction.

Figure 6-1 An ATP transaction



The amount of data that a requester application can send is limited to 578 bytes; the amount of data that a responder application can return is limited to 4624 bytes. The ATP programming interface includes a function that lets you add one or more single packets to follow the initial response, up to a total of eight packets including the initial number of packets sent, if you do not send eight packets in the initial response.

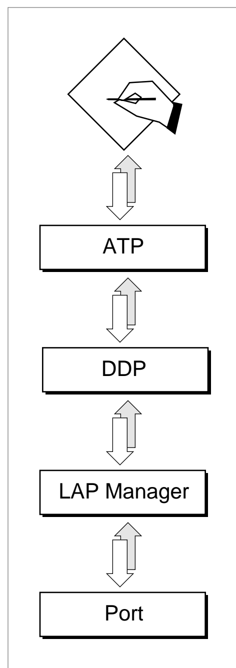
Note
Although you can use the ATP add-response function to extend the amount of response data, if you intend for your application to transfer large amounts of data, you should choose a transport protocol other than ATP. For example, you can use ADSP, which allows you to send and receive continuous streams of data. ◆

You can implement applications that use ATP to perform network-based transactions in the following two ways:

- You can write a single application that handles both the responder and requester actions of an ATP transaction and run that application on two networked nodes. This method allows each application to act as either the requester or the responder. The interaction remains asymmetric; only one side can control the communication during a single transaction. However, each side has the capacity to initiate a transaction by sending a request to the other side.
- You can write two distinct applications, one application that implements only the requester part of a transaction and another application that implements only the responder side. This scenario lends itself to a client-server model in which many nodes on a network run the requester application (client), while one or more nodes run the responder application (server); one server can respond to transaction requests from various clients.

ATP is a direct client of DDP, and it adds reliable delivery of data to the transport delivery services that DDP provides. Figure 6-2 shows ATP and the underlying protocol stack.

Figure 6-2 ATP and its underlying protocols



The ATP Packet Format

An ATP packet includes an 8-byte header followed by up to 578 bytes of data. An ATP packet is preceded by the DDP header that, in turn, is preceded by the data-link header, referred to as the *frame* header.

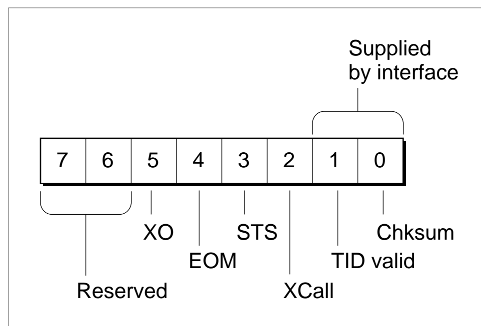
The ATP header contains the following information:

- The first byte consists of control information. Bits within this byte are set to identify aspects of a request or a response function.
- The second byte contains a ***bitmap/sequence number.*** This field is 8 bits wide, and its use and significance depend on whether the ATP packet is a request packet or a response packet. For request packets, this field is referred to as the ***transaction bitmap,*** and it identifies the number of buffers that a requester application has reserved for the response data. For response packets, this field is referred to as the ***ATP sequence number,*** and it is used to identify the sequential position of the response packet in the complete response message; ATP uses the sequence number to manage and handle lost or out-of-sequence response packets.

- The third and fourth bytes carry the transaction ID assigned to a request and used by the response to that request.
- The fifth through eighth bytes carry user data; an application can use these bytes for its own purposes, for example, to transfer command information.

The ATP data follows the header. It can consist of from 0 to 578 bytes. An ATP packet is enclosed in a DDP datagram that is enclosed in a data-link frame. Figure 6-3 shows a close-up view of the first byte of the ATP header, the control information byte.

Figure 6-3 The ATP packet header control information byte



The Control Information Byte

ATP applications call response and request functions that generate request and response packets. (ATP uses the release packet internally.) When set, the bits have the following meanings:

Bit	Meaning
0	Use the DDP checksum feature for this packet.
1	ATP has assigned the request transaction ID; the TID field value is now valid.
2	This request uses an extended parameter block.
3	To the requester: retransmit the request immediately (send-transmission status).
4	This is the last packet of the response message (end of message).
5	This request is an exactly-once transaction.

The Bitmap/Sequence Number

ATP ensures reliable delivery of data. This means that ATP retransmits all lost or dropped packets, and if it is unable to complete a transaction properly, ATP returns an error as the function result. To receive all the packets that make up a response message, a requester application must provide enough buffer space to hold the data. A request message consists of a single packet, while each ***response message*** can contain up to eight response packets.

Response packets are numbered from 0 to 7. ATP uses the sequence number to manage the transmission and receipt of response packets; the packet header ATP sequence number field contains 8 bits, 1 for each response packet.

ATP sets the sequence number in the request header to tell the .ATP driver code on the responder side which response packets the requester has not received. When a requester does not receive a complete response message, the .ATP driver code on the responder side can then send again only the packets that the requester side has not received, based on the bit settings of the transaction sequence number. ATP handles the retransmission of data internally without requiring any action on the part of your application. For information about the buffer records, see "The Buffer Data Structure" on page 6-20.

The Transaction ID

The third and fourth bytes of the ATP header carry a 16-bit transaction ID. The .ATP driver code on the requester side of a transaction assigns a unique transaction ID to each request that a requester application makes. The responder application that services the request includes this number as a parameter to the response call that it issues to send its response back to the requester. The transaction ID ties together the request and its response, ensuring that ATP delivers the correct data in response to each request. An application can issue and have pending multiple concurrent asynchronous requests; ATP uses the transaction ID to keep track of them.

User Bytes

ATP does not concern itself with the last 4 bytes of the ATP header. They are reserved for your use. You can use these bytes for any purpose prearranged by the requester and responder applications. The ATP functions include a parameter that you use to specify this data.

At-Least-Once and Exactly-Once Transactions

ATP supports two types of transactions: at-least-once transactions and exactly-once transactions. An ***at-least-once transaction*** ensures that the responder application receives every request directed to it at least once. However, this mode allows for the possibility of a responder application receiving duplicate requests.

For example, when you send a request that the .ATP driver code on the responder side receives, it passes the request on to the responder application. Your responder application then processes the request and creates a response to it. The ATP responder driver sends that response to your requester application. If the response is lost during the transmission, ATP retransmits the request after a period of time passes; you can set a value to control this timeout period. The ATP responder driver code receives the duplicate request and repeats the cycle of passing it on to your responder application for processing. At-least-once transactions ensure that the data is delivered at least once, and possibly more than once. You can use this transaction mode if it does not have adverse affects on the responder application.

An ***exactly-once transaction*** ensures that the responder application receives a specific request only once. These are also referred to as *XO,* as in *exactly-once* transactions. To create this result, the ATP responder code saves the response packets until the transaction is complete. This means that ATP itself can retransmit packets without requiring that your responder application reprocess the request.

The ATP responder code saves the response packets until the ATP code on the requester side indicates that it has received all of the packets. The ATP code on the requester side sends a transaction release packet to the ATP code on the responder side to signal that the requester application has received all of the response packets, so that ATP can now release them.

Because the transaction release packet could also be lost during transmission, ATP backs up this process with a transaction release timer. ATP marks packets saved for retransmission with a timestamp. When a packet ages beyond the amount of time that you set for the responder's release timer, ATP discards the packet.

You can set the release timer value that the ATP code on the responder end uses from your requester application; the send request functions include a release timer parameter for this purpose. For more information about this parameter, see "PSendRequest" on page 6-24 or "PNSendRequest" on page 6-27.

The Buffer Data Structure

The responder application needs to provide space to store the data to be sent to the requester until the requester application has received all of the data. The requester application needs to provide space to receive the data that it expects to receive as a result of the transaction. Each response can include up to eight packets. To handle the storage of these packets, the ATP client application at each end of the transaction provides a buffer data structure. The buffer data structure is designed to allow ATP to easily manage reliable transfer of multiple packets belonging to a single response message. A buffer data structure consists of an array of eight elements, each of which contains a pointer to a record of type `BDSElement`.

Each record contains a field for the size of the buffer created to hold the data and a pointer to that buffer. It also contains fields for the size of the data in the response packet and the user bytes that were passed in the packet header, if these bytes were used to communicate additional information. You can create your own buffer data structures, or you can use the ATP utility provided for this purpose. For a description of the BDS data type, see "The Buffer Data Structure" on page 6-20. For a description of the utility that you can use to build the buffer data structure, see "BuildBDS" on page 6-44.

ATP Flags

Many of the functions that you use for an ATP transaction pass control information in an ATP parameter block field called `atpFlags`. This field comprises a single byte whose bits you can set to signal control information, if appropriate. In some cases, ATP sets these flag bits for its own use. The discussion of each function that uses these flags includes the control information about the bits specific to that function. Table 6-1 shows the Pascal and assembly constants defined for these bits.

Table 6-1 Constants for ATP flag bits

Bit	Pascal constant	Assembly constant	Meaning
0	`atpSendChkvalue`	`sendChk`	Use DDP's checksum feature when sending a packet.
1	`atpTIDValidvalue`	`tidValid`	The transaction ID value that ATP assigns is set; you can check the `reqTID` field now.
2	None	`atpXcall`	This exactly-once transaction request uses an extended parameter block, the last field of which (`TRelTime`) is set to the release timer value for the ATP responder side.
3	`atpSTSvalue`	`atpSTSBit`	The ATP requester must retransmit a request immediately. (ATP sets the send-transmission-status bit, which it uses internally.)
4	`atpEOMvalue`	`atpEOMBit`	The last packet in this response is the end of the message.
5	`atpXOvalue`	`atpXOBit`	This request is an exactly-once transaction.

Using ATP

This section describes how to use ATP to

- send a transaction request to a responder application that is an ATP socket client
- receive a request from an ATP requester application and respond to it
- cancel pending ATP requests and responses

You can write a single ATP application that includes both the responder and requester code or two ATP applications that separately provide the responder and the requester services. This section describes how to write a requester application, and then it describes how to write a responder application.

Writing a Requester ATP Application

You use the `PSendRequest` function or the `PNSendRequest` function to send an ATP request to another socket.

Before you can use ATP, you must first open the .MPP driver, which in turn opens the .ATP driver. Use the Device Manager's `OpenDriver` function to open the .MPP driver. Even if you suspect that the .MPP and the .ATP drivers are open, you should call the `OpenDriver` function for the .MPP driver to ensure that this is the case. Calling `OpenDriver` for a driver that is already open will not produce harmful repercussions. See the chapter "Device Manager" in *Inside Macintosh: Devices* for information on the `OpenDriver` function. Do not close the .MPP driver when you are finished using

ATP because other applications dependent on it or on the .ATP driver require that it remain open.

To send an ATP request, follow these steps:

1. Create a buffer data structure (BDS) to hold the data that you expect to receive in response to your request. For information on how to do this, see "Creating a Buffer Data Structure" on page 6-12.
2. To allow ATP to assign the socket to be used to send the request, use the `PSendRequest` function. To specify a particular socket to be used to send the request, use the `PNSendRequest` function; in this case, you must call `POpenATPSocket` to first open the socket (see "POpenATPSkt" on page 6-30 for information about this function). For information on the parameters required for these functions, see "Specifying the Parameters for the Send Request Function" on page 6-12.
3. You can get the transaction ID that ATP assigns to a request from the `reqTID` parameter; you need this ID to cancel a request. However, before you check this field, make sure that the valid transaction ID (`atpTIDValidvalue`) bit (bit 1) of the `atpFlags` parameter is set. ATP sets this bit to inform you that it has assigned a transaction ID and that the `reqTID` field is now valid.
4. If you opened a socket to be used for the `PNSendRequest` call, close the socket using `PCloseATPSkt`. See"PCloseATPSkt" on page 6-31 for information on how to use this function. If you use the `PSendRequest` function, ATP allocates a socket and opens and closes it for you.

The code in Listing 6-1 shows how to open a socket and issue a call to the `PSendRequest` function. The code uses the `BuildBDS` function to create a buffer data structure to hold the response data it expects in response. This segment of code assumes that the application has already called the `OpenDriver` function to open the .MPP and .ATP drivers.

Listing 6-1 Opening a socket and sending an ATP request

```
CONST
   kMaxPacketSize = 578;            {maximum packet size we can receive}
   kNRespBuffs = 8;                 {you allow eight response buffers}
   kOurRespBufSize = kMaxPacketSize * kNRespBuffs;
                                    {response buffer size}
VAR
   err:            OSErr;
   reqLength:      Integer;
   nBufs:          Integer;
   ref:            Integer;
   targetAddr:     AddrBlock;
   gAtpPBPtr:      ATPPBPtr;
   gReqBufPtr:     Ptr;
   gRespBufPtr:    Ptr;
   gSRespBdsPtr:   BDSPtr;
```

```
BEGIN
   gAtpPBPtr := ATPPBPtr(NewPtr(SizeOf(ATPParamBlock)));
   gReqBufPtr := NewPtr(kMaxPacketSize);
   gRespBufPtr := NewPtr(kOurRespBufSize);
   gSRespBdsPtr := BDSPtr(NewPtr(SizeOf (BDSType)));
   err := OpenDriver('MPP',ref);
   if err <> noErr THEN DoErr(err);
   WITH gAtpPBPtr^ DO

   BEGIN
      atpSocket := 0;                    {dynamically allocate a socket}
      addrBlock.aNet := 0;               {accept requests from anyone}
      addrBlock.aNode := 0;
      addrBlock.aSocket := 0;
   END;
   err := POpenATPSkt(gAtpPBPtr,false);{socket is returned in }
                                       { gAtpPBPtr^.atpSocket}

   IF err <> noErr THEN DoErr(err);
   IF gAtpPBPtr^.ioResult <> noErr THEN DoErr(err);

   MyPrepareRequestData(gReqBufPtr,@reqLength);
                                       {user routine that prepares the }
                                       { request data to be sent}
   MyLocateTargetAddress(@targetAddr);
                                       {user routine that locates the }
                                       { target machine}

   {Set up your BDS structure.}
   nBufs := BuildBDS(gRespBufPtr,Ptr(gSRespBdsPtr),kOurRespBufSize);

   WITH gAtpPBPtr^ DO
   BEGIN
      atpFlags := atpXOvalue;            {issue an exactly-once transaction}
      addrBlock.aNet := targetAddr.aNet;
                                       {set up the target machine}
      addrBlock.aNode := targetAddr.aNode;
      addrBlock.aSocket := targetAddr.aSocket;

      reqLength := reqLength;            {size of your request data}
      reqPointer := gReqBufPtr;          {pointer to actual request data}
      numOfBuffs := nBufs;               {number of responses expected}
      bdsPointer := Ptr(gSRespBdsPtr); {your BDS pointer}
      timeOutVal := 3;                   {timeout interval}
      retryCount := 5;                   {number of retries}
   END;
   err := PSendRequest(gAtpPBPtr,false);
```

```
   IF err <> noErr THEN DoErr(err);

   MyProcessResponses(gAtpPBPtr^.bdsPointer,gAtpPBPtr^.numOfResps);
                                          {user routine to process the }
                                          { response data returned}

   {Clean up after you are done.}
   DisposePtr(Ptr(gAtpPBPtr));
   DisposePtr(gReqBufPtr);
   DisposePtr(gRespBufPtr);
   DisposePtr(Ptr(gSRespBdsPtr));
END.
```

Creating a Buffer Data Structure

Response data can comprise up to eight packets. ATP uses the organization of the buffer data structure (BDS) to manage these packets and ensure their complete delivery. The BDS must be an array of up to eight elements. You can create the buffer data structure yourself, or you can use the `BuildBDS` function for this purpose. You pass `BuildBDS` a pointer to a buffer and the length of the buffer, and it creates up to eight elements, one for each packet, depending on the size of the buffer that you supply. `BuildBDS` returns as its function result the number of elements that it creates; you pass this number and a pointer to the buffer data structure to the `PSendRequest` or `PNSendRequest` function that you call to issue the request. The memory that you allocate for the buffer must be nonrelocatable until the `PSendResponse` call completes execution. After `PSendResponse` returns, you should release this memory if you do not intend to reuse it.

Specifying the Parameters for the Send Request Function

When you call either the `PSendRequest` function or the `PNSendRequest` function to send an ATP request, you must do these tasks:

- Specify as the value of the `addrBlock` parameter the AppleTalk internet address of the socket whose client responder application you are sending the request to.
- Specify in the `reqLength` field the size in bytes of the request and in the `reqPointer` field a pointer to the request data. The buffer that you use to store the request belongs to ATP until the `PSendRequest` (or `PNSendRequest`) function completes execution, after which you can either reuse the memory or release it.
- Set the `timeOutVal` and `retryCount` parameters appropriately for your network. See the following section, "Setting the Timeout and Retry Count Parameters." If this is an exactly-once request, set bit 5 (`atpXOvalue`) of the `atpFlags` parameter to ensure that the responder application receives a specific request only once. For additional information about exactly-once transactions, see "At-Least-Once and Exactly-Once Transactions" on page 6-7.

You can send up to 4 bytes of additional information in the userData parameter, and ATP will pass this to the responder application in the userData parameter of its PGetRequest call. To make this parameter meaningful, both the requester and the responder applications should agree on the use of these additional data bytes that are separate from the request or response data sent in an ATP transaction.

Setting the Timeout and Retry Count Parameters

When a transaction does not complete on the first transmission, ATP retries it a number of times. You can control ATP's retry behavior by setting these two parameters: the timeOutVal field and the retryCount field. The timeOutVal value determines in seconds how long ATP waits before resending the original request packet; the retryCount value determines how many times ATP retries to send the request.

ATP optimizes how it performs retries based on the response bitmap; ATP on the requester side resends the request with the header bitmap indicating to the ATP driver on the responder side which packets it should resend. (See the "The Bitmap/Sequence Number" on page 6-6 for more information.) ATP makes this request to resend until it receives all of the packets or it exhausts the number of retry attempts that you specify. If ATP exhausts all of the retry attempts before the requester side receives all of the packets, ATP returns an error.

To choose the correct timeout value and retry count combination, you should consider the speed and complexity of your network—for example, take into account the degree of traffic congestion and whether your network contains multiple routers. You can use the AppleTalk Echo Protocol (AEP) echo socket to test the network performance and adjust the values accordingly. For more information about using the AEP echo socket to test network performance, see the chapter "Datagram Delivery Protocol (DDP)" in this book. You can store various pairs of values in a preferences resource file so that you can easily change them to adapt to the speed of the network.

If you want ATP to retry indefinitely to send the request, you can set the retryCount parameter to 255. In this case, ATP will send the request repeatedly until either the ATP responder end satisfies the request and sends back a response or you cancel the request. To cancel a PSendRequest call, you can use either the PKillSendReq function or the PRelTCB function. To cancel a PNSendRequest call, you can use the PKillSendReq function only.

Setting the Release Timer Value

For exactly-once transactions, the ATP responder code saves the response packets until the ATP code on the requester side indicates that it has received all of them. When this is the case, the ATP code on the requester side sends a transaction release packet to tell the ATP code on the responder side to release the response packets. Because this packet could be dropped or lost during transmission, ATP uses a release timer to discard the retained packets after a specified amount of time and to release the memory used to store them.

If the nodes at both ends of the ATP connection are running AppleTalk Phase 2 drivers, you can control the release timer value that determines when ATP releases the response packets by setting the 3 lower bits of the `TRelTime` parameter to one of the following values:

TRelTime	Setting of release timer
`000`	30 seconds
`001`	1 minute
`010`	4 minutes
`100`	8 minutes

Writing a Responder ATP Application

A responder application receives incoming ATP requests, processes them, and sends a response to the requester application. To write a responder application, you open a socket that you set up to listen for requests. When you receive a request, you process it and send a response back to the requester application. The response can consist of a message reporting the outcome of the processing you performed or data resulting from the processing.

Before you can use ATP, you must first open the .MPP driver, which in turn opens the .ATP driver. Use the Device Manager's `OpenDriver` function to open the .MPP driver. Even if you suspect that the .MPP and the .ATP drivers are open, you should call the `OpenDriver` function for the .MPP driver to ensure that this is the case. Calling `OpenDriver` for a driver that is already open will not produce harmful repercussions. See the chapter "Device Manager" in *Inside Macintosh: Devices* for information on the `OpenDriver` function. Do not close the .MPP driver when you are finished using ATP because other applications dependent on it or the .ATP driver require that it remain open.

Opening and Setting Up a Socket to Receive Requests

To open a socket to receive incoming requests, you use the following procedure:

1. To open the socket, call the `POpenATPSkt` function, providing it with values as follows:
   - □ To direct ATP to open a specific socket, provide the number of that socket as the value of the `atpSocket` parameter; to allow ATP to dynamically assign a socket, specify 0 as the value of this field.
   - □ To filter the sockets from which you will accept requests, set the internet socket address fields of the `addrBlock` parameter; to accept requests from any socket, set all three fields to 0. You can filter requests based on network, socket, or node numbers. For example, to accept requests from all sockets on the node whose ID is 112, you set the network and socket number fields of the address block record to 0 and the node ID field to 112.

2. To set up the socket to receive requests, call the `PGetRequest` function, which listens for an incoming request on the socket you specify. You provide it with the parameter values as follows:
   - Allocate a buffer to store the incoming request; you pass `PGetRequest` a pointer to this buffer and the length of the buffer. Unless you know the exact size of the incoming request, allocate at least 578 bytes of nonrelocatable memory for this buffer to accommodate the maximum request packet size. Set the `reqPointer` parameter to point to the buffer, and set the `reqLength` parameter to the size in bytes of the buffer.
   - Set the `atpSocket` parameter to the number of the socket to be used to listen for the request; this is the socket you opened through the `POpenATPSkt` call.
   - Set the `ioCompletion` parameter. In most cases, you should issue the `PGetRequest` call asynchronously so that your application can continue execution while `PGetRequest` listens for an incoming call; the `PGetRequest` function returns after it receives an incoming request or encounters an error condition. If you issue this call asynchronously, you must either specify a completion routine or set the `ioCompletion` parameter to `NIL`. If you use a completion routine, before it exits, your completion routine can call the `PGetRequest` function again to listen for the next incoming request. If you do not use a completion routine, you must poll the `ioResult` field for indication of an incoming request to determine when the function completes execution and whether an error condition or an incoming request caused the function to complete. For more information on calling a routine asynchronously, see the chapter "Introduction to AppleTalk" in this book.
3. Process the values that `PGetRequest` returns. The `PGetRequest` function returns the following values that may be of use to your application:
   - The request transaction ID `reqTID` that ATP assigns to this request. If you intend to respond to the request, save this value because you will need to pass it to the `PSendResponse` function and the `PAddResponse` function to identify the request for which the response message is intended. For more information on the transaction ID, see the discussion in the section "The ATP Packet Format" beginning on page 6-5.
   - The `userData` parameter, which contains any additional information that the requester application has sent. To make this parameter meaningful, both the requester and the responder applications should agree on the use of these additional data bytes that are separate from the request or response data sent in an ATP transaction.
   - The exactly-once bit (bit 5) of the `atpFlags` parameter, which is set if the request received is part of an exactly-once transaction. ATP uses this information internally to ensure that your responder application receives this request only once.

Listing 6-2 on page 6-17 shows how to open a socket and issue a call to the `PGetRequest` function to receive requests.

Responding to Requests

After you process a request and create a response message, you call the `PSendResponse` function to send the response. ATP assembles the response packets into a message and returns them to the requester application. You can send the request through the same socket that you use to receive incoming requests, or you can specify a different socket to be used for this purpose. To use a different socket, you must first open the socket by calling `POpenATPSocket`. The code in Listing 6-2 opens a new socket that it uses to send the response.

1. Create a buffer data structure to hold the response data that you want to send.

   The buffer data structure (BDS) must be an array of up to eight elements. You can use the `BuildBDS` function to create the BDS. You pass `BuildBDS` a pointer to a buffer and the length of the buffer, and it creates up to eight elements depending on the size of the buffer that you supply. `BuildBDS` returns as its function result the number of elements that it creates; you pass this number and a pointer to the buffer data structure to the `PSendResponse` call. The memory that you allocate for the buffer must be nonrelocatable until the `PSendResponse` call completes execution. After `PSendResponse` returns, you should release this memory.

2. To send the response, call the `PSendResponse` function. The response data cannot exceed 4624 bytes. If you need to send more information, you can follow the `PSendResponse` function with one or more calls to the `PAddResponse` function until you have sent a total of eight packets, including the packets that you sent when you called the `PSendResponse` function; each time you call the `PAddResponse` function, you can send one additional packet consisting of 578 bytes of data.

   - For the input address block (`addrBlock`) and transaction ID (`transID`) parameters to `PSendResponse`, use the address block (`addrBlock`) and request transaction ID (`reqTID`) parameter values that the `PGetRequest` function returned.
   - Set the `numOfBuffs` field to the number of response packets that you are sending. If you are sending fewer packets than the requester expects to receive, you must set the end-of-message (`atpEOMvalue`) bit (bit 4) in the `atpFlags` field to indicate that the last packet is the final one in the response message. The bitmap returned by the `PGetRequest` function indicates the number of packets that the requester expects in response.
   - Set the `atpSocket` field to the number of the socket that you are using to send the response.

3. Call the `CloseATPSkt` function to close the socket that you opened to receive requests and respond to them after you are finished with this socket. You can use the socket to continue to listen for requests until your application completes execution, but you should explicitly close the socket before exiting the program.

The code in Listing 6-2 first shows how to open a socket and issue a call to the `PGetRequest` function to receive requests. Then it shows how to prepare the response data and send it.

Listing 6-2 Opening a socket to receive a request and sending response data

```
CONST
   kMaxPacketSize = 578;             {maximum packet size you can receive}
   kMaxResponses = 8;                {maximum number of responses to expect}
   kRespBufSize = kMaxPacketSize * kMaxResponses;
                                     {your response buffer}
VAR
   err:                  OSErr;
   NumOfBufs:            Integer;
   ref:                  Integer;
   nBufs:                Integer;
   ReqBitMap:            BitMapType;
   thisBit:              LongInt;
   gAtpPBPtr:            ATPPBPtr;
   gSendRespPBPtr:       ATPPBPtr;
   gGetReqBufPtr:        Ptr;
   gSRespBuf:            Ptr;
   gSRespBdsPtr:         BDSPtr;
BEGIN
   gAtpPBPtr := ATPPBPtr(NewPtr(SizeOf(ATPParamBlock)));
   gSendRespPBPtr := ATPPBPtr(NewPtr(SizeOf(ATPParamBlock)));
   gGetReqBufPtr := NewPtr(kMaxPacketSize);
   gSRespBdsPtr := BDSPtr(NewPtr(SizeOf(BDSType)));
   gSRespBuf := NewPtr(kRespBufSize);

   err := OpenDriver('MPP',ref);
   if err <> noErr THEN DoErr(err);

WITH gAtpPBPtr^ DO
BEGIN
   atpSocket := 0;                   {dynamically allocate a socket}
   addrBlock.aNet := 0;              {accept requests from anyone}
   addrBlock.aNode := 0;
   addrBlock.aSocket := 0;
END;
err := POpenATPSkt(gAtpPBPtr,false);{socket is returned in }
                                    { gAtpPBPtr^.atpSocket}

IF err <> noErr THEN DoErr(err);
IF gAtpPBPtr^.ioResult <> noErr THEN DoErr(err);
```

```
WITH gAtpPBPtr^ DO
BEGIN
   reqLength := 0;                    {request data length will be returned }
                                      { to you here}
   reqPointer := gGetReqBufPtr;       {pointer to buffer for incoming request }
                                      { data}
END;
   err := PGetRequest(gAtpPBPtr,TRUE);{asynchronous PGetRequest}

   IF err <> noErr THEN DoErr(err);

   {Poll ioResult until the call completes.}
   WHILE gAtpPBPtr^.ioResult > noErr DO
   BEGIN
      GoDoSomething;                  {return control to user while you wait }
                                      { for PGetRequest to complete}
   END;

   IF gAtpPBPtr^.ioResult <> noErr THEN DoErr(err);

   MyProcessRequestReceived(gAtpPBPtr^.reqPointer,gAtpPBPtr^.reqLength)
                                      {user routine that looks at the request }
                                      { data received}

   {Walk through the bitmap and see how many response buffers you need.}
   NumOfBufs := 0;
   FOR thisBit := 0 to 7 DO
   BEGIN
      {Each bit that is set corresponds to a buffer.}
      if BitTst(@gAtpPBPtr^.bitMap,thisBit) = TRUE THEN
   BEGIN
      {Your routine to fill in the appropriate response data.}
      SetUpResponseData(gSRespBuf,thisBit);
      NumOfBufs := NumOfBufs + 1;
   END
END;

{Put your response data into the BDS structure.}
nBufs := BuildBDS(gSRespBuf,Ptr(gSRespBdsPtr),(NumOfBufs * kMaxPacketSize));
```

```
WITH gSendRespPBPtr^ DO
BEGIN
   atpSocket := gAtpPBPtr^.atpSocket;
   atpFlags := atpEOMvalue;         {indicate end of message}

{Send response to the machine that sent you the request.}
   addrBlock.aNet := gAtpPBPtr^.addrBlock.aNet;
   addrBlock.aNode := gAtpPBPtr^.addrBlock.aNode;
   addrBlock.aSocket := gAtpPBPtr^.addrBlock.aSocket;
   bdsPointer := Ptr(gSRespBdsPtr);
   numOfBuffs := NumOfBufs;         {send all of the responses back now}
   bdsSize := nBufs;                {indicate how many responses you are }
                                    { sending}
   transID := gAtpPBPtr^.transID;   {use transID returned from the }
                                    { PGetRequest function}
END;
err := PSendResponse(gSendRespPBPtr,FALSE);

IF err <> noErr THEN DoErr(err);

{Clean up after you are done.}
DisposePtr(Ptr(gAtpPBPtr));
DisposePtr(Ptr(gSendRespPBPtr));
DisposePtr(gGetReqBufPtr);
DisposePtr(Ptr(gSRespBdsPtr));
DisposePtr(gSRespBuf);
END.
```

Canceling an ATP Function

You can cancel all pending ATP function calls made on a specific socket by closing the socket. However, ATP provides functions that allow you to cancel individual function calls or all function calls of a particular kind. If you want to close a socket for which there are still pending requests that you don't want executed, you should first explicitly cancel those requests by using the ATP function provided for this purpose, instead of simply closing the socket.

You can use the following functions to cancel specific requests:

- To cancel a `PGetRequest` function, use the `PKillGetReq` function, which is described on page 6-41. You identify the request to be canceled by specifying the pointer to the parameter block that you passed to the `PGetRequest` function when you called it.

- To cancel all pending PGetRequest functions on a certain socket, use the ATPKillAllGetReq function described on page 6-42; you specify the socket number, whose pending get requests you want to cancel, as the value of the atpSocket parameter.
- To cancel a PSendRequest or a PNSendRequest function, use the PKillSendReq function described beginning on page 6-38. You identify the request to be canceled by specifying the pointer to the parameter block that you passed to the function when you issued it. To cancel a PSendRequest function, use the PRelTCB function described beginning on page 6-40. You identify the request to be canceled by specifying the request transaction ID as the transID parameter and the destination socket of the request as the addrBlock parameter.
- To cancel an exactly-once PSendResponse function, use the PRelRspCB function, described beginning on page 6-43. You identify the request to be canceled by specifying the transaction ID of the associated request as the transID parameter and the PSendResponse destination socket number as the atpSocket parameter.

ATP Reference

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

- The "Data Structures" section shows the Pascal data structures for the buffer data structure (BDS) array, the ATP parameter block, and the address block record.
- The "Routines" section describes the ATP routines for making a transaction request, receiving and responding to a transaction request, canceling a call to an ATP function, and building a buffer data structure to be used to hold response data to be sent and received.

Data Structures

This section describes the data structures that are specific to ATP. These data structures include the buffer data structure that is used to hold the response data packets to be sent from one application and received by another, the ATP parameter block that is used to hold input and output values for ATP functions, and the address block record data structure that ATP functions use to specify an AppleTalk internet socket address.

The Buffer Data Structure

The buffer data structure (BDS) is an array of type BDSElement containing up to eight records, each of which is used to hold a response packet. You create a BDS to hold the response data that you send using the PSendResponse function. You also create a BDS to receive the response packets that you request through a PSendRequest or PNSendRequest function. You can use the BuildBDS function to create this data structure, or you can create the data structure in Pascal.

```
TYPE  BDSElement =
RECORD
   buffSize: Integer;
   buffPtr: Ptr;
   dataSize: Integer;
   userBytes: Longint;
END;
BDSType = ARRAY[0..7] OF BDSElement;
BDSPtr = ^BDSType;
BitMapType = PACKED ARRAY[0..7] OF Boolean;
```

Field descriptions

`buffSize`	The size in bytes of the buffer.
`buffPtr`	A pointer to the buffer.
`dataSize`	The size of the data received.
`userBytes`	Up to 4 bytes of additional data separate from the response data.

The ATP Parameter Block

The ATP functions require a pointer to an ATP parameter block that is used to pass the input and output parameters associated with the function. The `ATPParamBlock` data type defines the ATP parameter block. The ATP parameter block includes variant records for the fields that are particular to an ATP routine.

This section defines the fields that are common to all ATP functions that use the ATP parameter block. (The `BuildBDS` function does not use the ATP parameter block.) These common fields are either filled in by the MPW interface or returned by the function; your application does not need to provide values for these fields. This section does not define reserved fields, which are used internally by the .ATP driver or not at all. The fields that are used for specific functions only are defined in the descriptions of the functions to which they apply.

```
TYPE ATPParamBlock =
PACKED RECORD
   qLink:         QElemPtr;          {reserved}
   qType:         Integer;           {reserved}
   ioTrap:        Integer;           {reserved}
   ioCmdAddr:     Ptr;               {reserved}
   ioCompletion:  ProcPtr;           {completion routine}
   ioResult:      OSErr;             {result code}
   userData:      Longint;           {ATP user bytes}
   reqTID:        Integer;           {request transaction ID}
   ioRefNum:      Integer;           {driver reference number}
   csCode:        Integer;           {call command code}
   atpSocket:     Byte;              {currBitMap or socket number}
```

```
   CASE MPPParmType OF
      SendRequestParm,
      SendResponseParm,
      GetRequestParm,
      AddResponseParm,
      KillSendReqParm:
         (atpFlags:         Byte;    {control information}
         addrBlock:         AddrBlock;
                                     {source/dest. socket address}
         reqLength:         Integer; {request/response length}
         reqPointer:        Ptr;     {ptr to request/response data}
         bdsPointer:        Ptr;     {ptr to response BDS}
      CASE MPPParmType OF
         SendRequestParm:
            (numOfBuffs:    Byte;    {number of responses expected}
            timeOutVal:     Byte;    {timeout interval}
            numOfResps:     Byte;    {number of responses }
                                     { actually received}
            retryCount:     Byte;    {number of retries}
            intBuff:        Integer; {used internally for }
                                     { PNSendRequest}
            TRelTime:       Byte);   {TRelease time for extended }
                                     { send request}
         SendResponseParm:
            (filler0:       Byte;    {bitmap}
            bdsSize:        Byte;    {number of BDS elements}
            transID:        Integer);{transaction ID}
         GetRequestParm:
            (bitmap:        Byte;    {bitmap}
            filler1:        Byte);   {reserved}
         AddResponseParm:
            (rspNum:        Byte;    {sequence number}
            filler2:        Byte);   {reserved}
         KillSendReqParm
            (aKillQEl:      Ptr));   {ptr to (queue element) function to }
                                     { cancel}
END;

ATPPBPtr =  ^ATPParamBlock;
```

Field descriptions

`ioCompletion` A pointer to a completion routine that you can provide. When you execute a function asynchronously, the .ATP driver calls your completion routine when it completes execution of the function if

	you specify a pointer to the routine as the value of this field. Specify `NIL` for this field if you do not wish to provide a completion routine. If you execute a function synchronously, the .ATP driver 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 function asynchronously, the .ATP driver sets this field to 1 as soon as you call the function, and it changes the field to the actual result code when the function completes execution.
`ioRefNum`	The .ATP driver reference number. The MPW interface fills in this field.
`csCode`	The command code for the ATP function to be executed. The MPW interface fills in this value for you.

The Address Block Record

The address block record defines a data structure of `AddrBlock` type. The following ATP functions use this data type to specify AppleTalk internet socket addresses: `PSendRequest`, `PSendResponse`, `PNSendResponse`, `POpenATPSkt`, `PGetRequest`, `PSendResponse`, `PAddResponse`, `PRelTCB`, `PRelRspCB`.

```
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 ATP client application whose address you are specifying.
`aNode`	The node ID of the machine running the ATP client application whose address you are specifying.
`aSocket`	The number of the socket used for the ATP client application.

Routines

This section describes the ATP routines that you use to

- send a request to a responder socket client
- open and close an ATP socket
- set up a socket to listen for a request
- send a response to a requester socket client
- cancel a response or a request function
- build a buffer data structure to store the response data

All of the ATP functions except the `BuildBDS` function use the ATP parameter block to pass input and output parameters. Each function description shows the parameter block for that function. An arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both:

Arrow	Meaning
→	Input
←	Output
↔	Both

Sending an ATP Request

This section describes the `PSendRequest` function that you use to send a request to another socket's client application, allowing ATP to dynamically allocate the socket to be used to send the request; in this case, ATP opens the socket when you issue the function and closes it after the call completes execution. It also describes the `PNSendRequest` function that you can use to send a request to another socket while specifying the socket to be used to send the request; you must open the socket to be used and close it when you're finished with it.

PSendRequest

The `PSendRequest` function sends a request to another socket whose client application is to respond to the request. `PSendRequest` then waits for a response before completing execution.

```
FUNCTION PSendRequest (thePBPtr: ATPPBPt; async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	`iocompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`userData`	`LongInt`	Four bytes of user data.
←	`reqTID`	`Integer`	The transaction ID for this request.
→	`csCode`	`Integer`	Always `sendRequest` for this function.
←	`currBitMap`	`Byte`	A bitmap.
↔	`atpFlags`	`Byte`	The control information.
→	`addrBlock`	`AddrBlock`	The destination socket address.
→	`reqLength`	`Integer`	The size in bytes of the request.

→	reqPointer	Ptr	A pointer to request data.
→	bdsPointer	Ptr	A pointer to response data.
→	numOfBuffs	Byte	The number of responses expected.
→	timeOutVal	Byte	The timeout interval.
←	numOfResps	Byte	The number of responses received.
↔	retryCount	Byte	The number of retries.
→	TRelTime	Byte	The release timer setting.

Field descriptions

`userData` Four bytes of user data that are sent in the header of the message. You can use these bytes for any purpose that you wish.

`reqTID` A number that identifies this transaction request. If you want to use the `PRelTCB` function to cancel the transaction, you must pass it this number.

`currBitMap` A bitmap showing which packets of the transaction were received.

`atpFlags` A control information field whose bits, numbered 0–7, are used as flags.

You set bit 5 (`atpXOvalue`) to specify an exactly-once transaction. To specify an at-least-once transaction, you clear the bit.

To set the other connection end's release timer, set bit 2 of this flag, and use the `TRelTime` field to indicate the amount of time. Bit 2 (`atpXcallvalue`) indicates that the parameter block is extended to include the release timer field.

ATP sets the `atpTIDValidvalue` bit (bit 1) of this field to indicate that the transaction ID field (`reqTID`) now contains valid data; you should determine if this bit is set before you check the request transaction ID.

To direct ATP to use DDP's checksum feature, set the send checksum (`atpSendChkvalue`) bit (bit 0) of this flag.

`addrBlock` The AppleTalk internet address of the socket to which the request is to be sent.

`reqLength` The size of the request to be sent.

`reqPointer` A pointer to the request data to be sent.

`bdsPointer` A pointer to a buffer data structure (BDS) that is to be used to hold the responses.

`numOfBuffs` On input, the number of response packets that you expect from the responder application. If this field contains a nonzero number on return, you can examine the `currBitMap` field to determine which packets of the transaction were actually received.

`timeOutVal` The number of seconds that ATP should wait for a response before resending the request.

`numOfResps` The number of responses actually received.

`retryCount` The maximum number of times ATP should retry to send the request. This field is used to monitor the number of retries; for each retry, ATP decrements it by 1.

`TRelTime` The release timer setting. Set the 3 lower bits of this field value to indicate the time to which the release timer should be set for the other end of the connection:

TRelTime	Setting of release timer
000	30 seconds
001	1 minute
010	4 minutes
100	8 minutes

DESCRIPTION

The `PSendRequest` function sends your request data to the destination ATP socket that you specify, and then it waits for that socket's client to return a response message. ATP dynamically assigns and opens the socket to be used to send the request, and it closes the socket when the function completes execution. Before you call the `PSendRequest` function, you must build a buffer data structure to hold the response data. You can use the `BuildBDS` function to do this. See "The Buffer Data Structure" on page 6-8 and "BuildBDS" on page 6-44 for a discussion of this function.

If you want to include additional information along with the request message, you can use the user bytes to include it; for example, you can use these bytes for command information.

The `PSendRequest` function completes execution when it receives an entire response or when the retry count is exceeded. The timeout value (`timeOutVal`) determines how many seconds `PSendRequest` waits before resending the original request packet. The retry count (`retryCount`) value determines the maximum number of times that ATP is to resend the request. Together the timeout value and the retry count determine the total retry time in seconds (`timeOutVal` x `retryCount` = total retry time). ATP modifies the retry count field value during execution of the `PSendRequest` function if it resends the request; ATP decrements the field by 1 for each retry. See "Writing a Requester ATP Application" beginning on page 6-9 for information on how to select these values.

The .ATP driver maintains a timer, called the *release timer,* for each call to the `PSendResponse` function that is part of an exactly-once (XO) transaction. If the timer expires before the transaction is complete (that is, before the socket receives the transaction release packet), the driver completes the `PSendResponse` function. Before AppleTalk Phase 2, the release timer was always set to 30 seconds. You can set the responding socket's release timer to a value other than 30 seconds. To do this, set the extended call bit (bit 2) of the `atpFlags` field in the parameter block for the `PSendRequest` function and specify the release timer parameter as the value of the

TRelTime field. The nodes at both ends of the ATP connection must be running AppleTalk Phase 2 drivers for this feature to work. For a discussion of exactly-once transactions and use of the release timer, see "At-Least-Once and Exactly-Once Transactions" on page 6-7. You should set the exactly-once flag (bit 5) if you want the request to be part of an exactly-once transaction.

You can use the PKillSendReq function or the PRelTCB function to cancel a PSendRequest call. For the PRelTCB function, you need the request transaction ID that ATP returns in the request transaction ID (reqTID) field of the PSendRequest call's parameter block. You can examine the request transaction ID field before the completion of the call, but its contents are valid only after the tidValid bit (bit 1) of the atpFlags field has been set. You should determine if this bit is set before you check the request transaction ID.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PSendRequest function from assembly language, call the _Control trap macro with a value of sendRequest in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
reqFailed	–1096	Retry count exceeded
tooManyReqs	–1097	Too many concurrent requests
noDataArea	–1104	Too many outstanding ATP calls
reqAborted	–1105	Request canceled

PNSendRequest

The PNSendRequest function sends a request to another socket's client. It uses the socket that you specify to send the request.

```
FUNCTION PNSendRequest (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

thePBPtr — A pointer to an ATP parameter block.

async — A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	iocompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	userData	LongInt	Four bytes of user data.
←	reqTID	Integer	The transaction ID for this request.
→	csCode	Integer	Always nSendRequest for this function.
→	atpSocket	Byte	The socket number to send the request.
↔	atpFlags	Byte	The control information.
→	addrBlock	AddrBlock	The destination socket address.
→	reqLength	Integer	The size in bytes of the request.
→	reqPointer	Ptr	A pointer to the request data.
→	bdsPointer	Pointer	A pointer to the BDS.
→	numOfBuffs	Byte	The number of responses expected.
→	timeOutVal	Byte	The timeout interval.
←	numOfResps	Byte	The number of responses received.
↔	retryCount	Byte	The number of retries.
←	intBuff	Integer	A buffer that ATP uses internally.
→	TRelTime	Byte	The release timer setting.

Field descriptions

`userData` Four bytes of user data that are sent in the header of the message. You can use these bytes for any purpose that you wish.

`reqTID` A number that identifies this transaction request.

`atpSocket` The socket to be used to send the request. You must have previously opened this socket by calling the `POpenATPSkt` function.

`atpFlags` A control information field whose bits, numbered 0–7, are used as flags.

You set bit 5 (`atpXOvalue`) to specify an exactly-once transaction. To specify an at-least-once transaction, you clear the bit.

To set the other connection end's release timer, set bit 2 of this flag (`atpXcallvalue`) to signal that this is an extended call and that the parameter block includes an additional field. Then you use the `TRelTime` field to indicate the amount of time.

ATP sets the `atpTIDidValidvalue` bit (bit 1) of this field to indicate that the transaction ID field (`reqTID`) now contains valid data; you should determine if this bit is set before you check the request transaction ID.

To direct ATP to use DDP's checksum feature, set the `atpSendChkvalue` bit (bit 0) of this flag.

`addrBlock` The AppleTalk internet socket address of the application to which the request is being sent.

`reqLength` The size in bytes of the request data to be sent.

`reqPointer` A pointer to the request data to be sent.

`bdsPointer` A pointer to the buffer data structure (BDS) that is to hold the data returned in response to the request.

`numOfBuffs` The number of response packets requested and expected from the responder application.

`timeOutVal`	The number of seconds that ATP should wait for a response before resending the request.
`numOfResps`	The number of response packets actually received.
`retryCount`	The maximum number of times ATP should retry to send the request. This field value is used to monitor the number of retries; for each retry, ATP decrements the value by 1.
`intBuff`	Two bytes that are used internally by ATP.
`TRelTime`	The release timer setting. The 3 lower bits of this field value indicate the time to which the release timer is to be set, as follows:

TRelTime	**Setting of release timer**
000	30 seconds
001	1 minute
010	4 minutes
100	8 minutes

DESCRIPTION

The `PNSendRequest` function is similar to the `PSendRequest` function except that rather than relying on ATP to dynamically allocate a socket to use for the transaction, `PNSendRequest` lets you specify the socket to be used to send the request. You set the `atpSocket` field of the parameter block to the number of the socket to be used for the request; you must have previously opened the socket by calling the `POpenATPSkt` function. `POpenATPSkt` lets you send more than one asynchronous request using the same socket. The number of concurrent requests that you send using `PNSendRequest` is machine dependent. If you exceed this limit, ATP returns an error message (`tooManyReqs`) indicating this. Note that if you call the `PNSendRequest` function without having previously opened the socket that you specify for the send request, ATP returns a bad ATP socket (`badATPSkt`) error.

The .ATP driver maintains a timer, called the *release timer,* for each call to the `PSendResponse` function that is part of an exactly-once (XO) transaction. If the timer expires before the transaction is complete (that is, before the socket receives the transaction release packet), the driver completes the `PSendResponse` function. Before AppleTalk Phase 2, the release timer was always set to 30 seconds. To set the other connection end's release timer to another value, set bit 2 of the `atpFlags` field in the parameter block for the `PNSendRequest` function to indicate that this is an extended call, then set the `TRelTime` field to the new value. The nodes at both ends of the ATP connection must be running AppleTalk Phase 2 drivers for this feature to work. For a discussion of exactly-once transactions and use of the release timer, see "At-Least-Once and Exactly-Once Transactions" on page 6-7. You should set the exactly-once flag if you want the request to be part of an exactly-once transaction.

You can use the `PKillSendReq` function to cancel a pending `PNSendRequest` call. Unlike `PSendRequest`, you cannot use the `PRelTCB` function to kill this request call.

SPECIAL CONSIDERATIONS

The parameter block for the PNSendRequest function requires 2 additional bytes, intBuff, for ATP's internal use. You must not modify these bytes.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PNSendRequest function from assembly language, call the _Control trap macro with a value of nSendRequest in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
reqFailed	–1096	Retry count exceeded
tooManyReqs	–1097	Too many concurrent requests
badATPSkt	–1099	Specified socket is not opened
noDataArea	–1104	Too many outstanding ATP calls
reqAborted	–1105	Request canceled

Opening and Closing an ATP Socket

This section describes the POpenATPSkt function that you use to open a socket for receiving ATP requests from another socket's client application. It also describes the PCloseATPSkt function that you use to close a socket used for receiving requests after you are finished with that socket. You also use the POpenATPSkt and PCloseATPSkt functions to open and close a socket that you want to use to send requests through a specific socket by calling the PNSendRequest function.

POpenATPSkt

The POpenATPSkt function opens a socket to be used to receive ATP requests or to be used to send ATP requests through the PNSendRequest function.

```
FUNCTION POpenATPSkt (thePBptr: ATPPBPtr; async: Boolean): OSErr;
```

thePBptr — A pointer to an ATP parameter block.

async — A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	csCode	Integer	Always openATPSkt for this function.
↔	atpSocket	Byte	The socket number to be used.
→	addrBlock	AddrBlock	The socket request specification.

Field descriptions

atpSocket — The number of the socket that ATP is to open. To direct ATP to dynamically assign a socket number, which it returns as the value of this field, specify 0.

addrBlock — A value that specifies the AppleTalk internet socket addresses that the atpSocket field will receive requests from; specify 0 for the network number, the node ID, or the socket number to accept all requests based on the value of that part of the AppleTalk internet socket address.

DESCRIPTION

The POpenATPSkt routine serves two purposes: you use it to open a socket to be used for incoming requests, and you use it to open a socket to send requests using a specific socket. (The PNSendRequest function lets you send a request using a specific socket, but you must first open that socket using POpenATPSkt.) You can use the addrBlock field to filter requests that you will accept by restricting network addresses.

ASSEMBLY-LANGUAGE INFORMATION

To execute the POpenATPSkt function from assembly language, call the _Control trap macro with a value of openATPSkt in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
tooManySkts	–1098	Too many responding sockets
noDataArea	–1104	Too many outstanding ATP calls

SEE ALSO

The PNSendRequest function is described on page 6-27.

PCloseATPSkt

The PCloseATPSkt function closes a socket that was opened to receive ATP requests or to send requests over a specific socket.

```
FUNCTION PCloseATPSkt (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

thePBPtr — A pointer to an ATP parameter block.

async — A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	csCode	Integer	Always closeATPSkt for this function.
→	atpSocket	Byte	The socket number.

Field descriptions

atpSocket — The number of the socket to be closed.

DESCRIPTION

The PCloseATPSkt function closes the socket that you opened to receive ATP requests or to send them over a specific socket.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PCloseATPSkt function from assembly language, call the _Control trap macro with a value of closeATPSkt in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
noDataArea	–1104	Too many outstanding ATP calls

Setting Up a Socket to Listen for Requests

After you open a socket to be used to response to requests, you need to set up that socket to receive requests. You use the PGetRequest function for this purpose.

PGetRequest

The PGetRequest function sets up a socket to listen for a request from another socket.

```
FUNCTION PGetRequest (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

thePBPtr — A pointer to an ATP parameter block.

async — A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
←	userData	LongInt	Four bytes of user data.
←	reqTID	Word	The transaction ID.
→	csCode	Integer	Always getRequest for this function.
→	atpSocket	Byte	The socket number.
←	atpFlags	Byte	The control information.
←	addrBlock	LongInt	The destination socket address.
↔	reqLength	Word	On input, the request buffer size. On return, the actual of the request received.
→	reqPointer	Ptr	A pointer to the request buffer.
←	bitMap	Byte	A bitmap.

Field descriptions

`userData` The 4 user bytes from the request.

`reqTID` The transaction ID of the request that `PGetRequest` has received. ATP supplies this value.

`atpSocket` The number of the socket that is to be used to listen for requests. This is the number of a socket you opened using the `POpenATPSkt` function call.

`atpFlags` A control information field whose bits, numbered 0–7, are used as flags.

ATP sets bit 5, the exactly-once flag (`atpXOvalue`), if the request received is part of an exactly-once transaction.

`addrBlock` The AppleTalk internet address of the socket from which the request was sent. ATP returns this value.

`reqLength` On input, the size in bytes of the buffer to be used to store the incoming request. On return, the actual number of bytes of the request received.

`reqPointer` A pointer to the location of the buffer to be used to store the incoming request.

`bitMap` A bitmap of the transaction that ATP returns.

DESCRIPTION

To receive an ATP request, you must set up a socket to listen for incoming requests; you use the `PGetRequest` function to do this. In almost all cases, you should call the `PGetRequest` function asynchronously to avoid delaying execution of your program until after an ATP request comes in. The `PGetRequest` function completes execution after it receives an ATP request.

The `PGetRequest` function returns the transaction ID of the request that it receives in the `reqTID` field. You should save this value if you intend to respond to the request; this transaction ID is used as an input parameter to the `PSendResponse` and `PAddResponse` functions. To determine that the request transaction ID specified in the `reqTID` field is valid, first check the `atpTIDValidvalue` bit (bit 1) of the `atpFlags` field. If this bit is set, the `reqTID` field value is valid.

You must allocate nonrelocatable memory to be used as the buffer to hold an incoming request. Make sure that you allocate enough memory to hold the entire request; ATP will not deliver more data than will fit in the amount of buffer space that you specified as the value of the `reqLength` field. The buffer should be 578 bytes long, which is the maximum size of a request packet, unless you know the exact size of the request.

SPECIAL CONSIDERATIONS

Memory used for the incoming request buffer belongs to ATP for the life of the call.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PGetRequest` function from assembly language, call the `_Control` trap macro with a value of `getRequest` in the `csCode` field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

`noErr`	0	No error
`badATPSkt`	–1099	Bad responding socket

SEE ALSO

For information on opening a socket that you can set up to receive requests, use the `POpenATPSkt` function, described on page 6-30.

Responding to Requests

After you receive and process a request, you can call the `PSendResponse` function to send the response data to the requesting socket. If you need to send additional data, you can call the `PAddResponse` function after you call `PSendResponse`. This section discusses the `PSendResponse` and `PAddResponse` functions.

PSendResponse

The `PSendResponse` function sends the response message to the requester.

```
FUNCTION PSendResponse (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	userData	LongInt	Four bytes of user data.
→	csCode	Integer	Always sendResponse for this function.
→	atpSocket	Byte	The socket number.
→	atpFlags	Byte	The control information.
→	addrBlock	AddrBlock	The destination socket address.
→	bdsPointer	Ptr	A pointer to the response BDS.
→	numOfBuffs	Byte	The number of response packets to be sent.
→	bdsSize	Byte	The BDS size in elements.
→	transID	Integer	The transaction ID.

Field descriptions

`userData` Four bytes of user data that are sent in the header of the message. If the response was part of an exactly-once transaction, this field contains the user bytes from the `TRel` packet.

`atpSocket` The number of the socket that is sending the response.

`atpFlags` A control information field whose bits, numbered 0–7, are used as flags.

To signal that this packet is the last packet in the transaction's response message when the number of responses is less than expected, set the end-of-message (`atpEOMvalue`) bit (bit 4).

ATP sets the send-transmission-status (`atpSTSvalue`) bit (bit 3) to force the requester to retransmit a request immediately, when this is necessary.

To direct ATP to use DDP's checksum feature, set the send checksum (`atpSendChkvalue`) bit (bit 0) of this flag.

`addrBlock` The AppleTalk internet socket address of the socket to which the response is to be sent.

`bdsPointer` A pointer to the response buffer data structure (BDS) that contains the response data.

`numOfBuffs` The number of response packets to be sent.

`bdsSize` The number of elements in the buffer data structure (BDS).

`transID` The transaction ID of the request for which this response is meant.

DESCRIPTION

You call `PSendResponse` when you receive a request, and after you have created a response message. The `PSendResponse` function sends the data to the socket whose address you specify; this is the address of the requester socket. If you cannot or do not want to send the entire response at one time, you can call `PSendResponse` to send the first part of it, then call `PAddResponse` later to send the remainder of the response.

To signal the requester socket that you are sending fewer response packets than it expects to receive, you must set the end-of-message flag (bit 4) of the `atpFlags` parameter.

For each call to the `PSendResponse` function that is part of an exactly-once (XO) transaction, ATP maintains a timer, called the *release timer.* If the timer expires before the transaction is completed, that is, before the socket receives the transaction release packet, ATP completes the `PSendResponse` function. Before AppleTalk Phase 2, the release timer was always set to 30 seconds. The `PSendRequest` or the `PNSendRequest` function can set the release timer for the responder to a different value. For more information about sending a response, see "Responding to Requests" beginning on page 6-16.

SPECIAL CONSIDERATIONS

During exactly-once transactions, `PSendResponse` doesn't complete until either a `TRel` packet is received from the socket that made the request or the retry count is exceeded.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PSendResponse` function from assembly language, call the `_Control` trap macro with a value of `sendResponse` in the `csCode` field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

`noErr`	0	No error
`badATPSkt`	–1099	Bad responding socket
`badBuffNum`	–1100	Sequence number out of range
`noRelErr`	–1101	No release received
`noDataArea`	–1104	Too many outstanding ATP calls

SEE ALSO

See the chapter "Introduction to AppleTalk" in this book for a description of the AppleTalk internet socket address structure.

For a description of the possible release timer values that `PSendRequest` or `PNSendRequest` can set, see either the `PSendRequest` function on page 6-24 or the `PNSendRequest` function on page 6-27.

PAddResponse

The `PAddResponse` function sends an additional response packet to a socket that has already been sent the first part of the response message through the `PSendResponse` function.

```
FUNCTION PAddResponse (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	userData	LongInt	Four bytes of user data.
→	csCode	Integer	Always addResponse for this function.
→	atpSocket	Byte	The source socket number.
→	atpFlags	Byte	The control information.
→	addrBlock	AddrBlock	The destination socket address.
→	reqLength	Integer	The size in bytes of the response data.
→	reqPointer	Ptr	A pointer to the response data.
→	rspNum	Byte	The sequence number.
→	transID	Integer	The transaction ID.

Field descriptions

`userData` Four bytes of user data that are sent in the header of the message. You can use these bytes for any purpose that you wish.

`atpSocket` The number of the socket that is used to send the additional response.

`atpFlags` A control information field whose bits, numbered 0–7, are used as flags.

To signal that this packet is the last packet in the transaction's response message when the number of responses is less than expected, set the end-of-message (`atpEOMvalue`) bit (bit 4).

ATP sets the send-transmission-status (`atpSTSvalue`) bit (bit 3) to force the requester to retransmit a request immediately, when this is necessary.

To direct ATP to use DDP's checksum feature, set the send checksum (`atpSendChkvalue`) bit (bit 0) of this flag.

`addrBlock` The number of the socket to which the additional response packet is to be sent.

`reqLength` The size in bytes of the response data to be sent.

`reqPointer` A pointer to the response data to be sent.

`rspNum` The sequence number of the response, in the range of 0 to 7.

`reqTID` The transaction ID of the request for which this response is meant.

DESCRIPTION

The `PAddResponse` function sends an additional response packet, following the initial response sent in return to a `PSendResponse` request message. You can send multiple additional response packets, one at a time, up to a total of eight packets including the initial response packets sent in the `PSendResponse` function.

You cannot issue a PAddResponse call without having first called PSendResponse. You must provide a pointer to the buffer containing the data to be sent and specify the amount of data. Each packet can contain up to 578 bytes of data. You also must specify the sequence number of the response.

SPECIAL CONSIDERATIONS

If the transaction is part of an exactly-once transaction, you must allocate nonrelocatable memory for the buffer that you use for the response data, and you must not alter the contents of this buffer until the corresponding PSendRequest function has completed execution.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PAddResponse function from assembly language, call the _Control trap macro with a value of addResponse in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
badATPSkt	–1099	Bad responding socket
badBuffNum	–1100	Sequence number out of range
noSendResp	–1103	PAddResponse issued before PSendResponse
noDataArea	–1104	Too many outstanding ATP calls

Canceling Pending ATP Functions

This section describes the functions that you use to cancel pending ATP functions. It describes the PKillSendReq function that you use to cancel a PSendRequest or PNSendRequest function, the PRelTCB function that you use to cancel a PSendRequest function, the PKillGetReq function that you use to cancel a PGetRequest function, the ATPKillAllGetReq function that you use to cancel all pending PGetRequest functions, and the PRelRspCB function that you use to cancel a PSendResponse call that specifies an exactly-once transaction.

PKillSendReq

The PKillSendReq function cancels the pending PSendRequest or PNSendRequest functions whose queue element pointer you specify.

```
FUNCTION PKillSendReq (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

thePBPtr	A pointer to an ATP parameter block.
async	A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to the completion routine.
←	ioResult	OSErr	The function result.
→	csCode	Integer	Always killSendReq for this function.
→	aKillQEl	Ptr	A pointer to queue element of function to be removed.

Field descriptions

aKillQEl	A pointer to the queue element of the pending function that is to be canceled. This is the pointer to the parameter block that you passed to the send request function when you issued the function.

DESCRIPTION

To cancel a specific pending PSendRequest or PNSendRequest function, you specify the pointer to the queue element for the function in the aKillQEl field of the parameter block for the PKillSendReq function, then call the function. If the function has already completed execution or if it is not in the ATP queue for any other reason, PKillSendReq returns a message (cbNotFound) indicating that it could not find the parameter block.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PKillSendReq function from assembly language, call the _Control trap macro with a value of killSendReq in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
cbNotFound	–1102	The aKillQEl parameter does not point to a PSendRequest or PNSendRequest queue element

SEE ALSO

To send requests, use the PSendRequest function, described on page 6-24, and the PNSendRequest function, described on page 6-27.

PRelTCB

The PRelTCB function cancels the pending PSendRequest function that you specify.

```
FUNCTION PRelTCB (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

thePBPtr	A pointer to an ATP parameter block.
async	A Boolean that indicates whether the function should be executed asynchronously or synchronously. Specify TRUE for asynchronous execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The function result.
→	csCode	Integer	Always relTCB for this function.
→	addrBlock	AddrBlock	The destination socket address.
→	transID	Integer	The transaction ID of the request to be canceled.

Field descriptions

addrBlock	The AppleTalk internet address of the destination socket for which the PSendRequest function that is to be canceled was meant.
transID	The transaction ID of the PSendRequest function to be canceled. You can get the transaction ID from the reqTID field of the PSendRequest parameter block queue entry.

DESCRIPTION

The PRelTCB function releases the queued parameter block for the PSendRequest function whose transaction ID you specify. The PRelTCB function returns a function result of reqAborted for the canceled PSendRequest function.

SPECIAL CONSIDERATIONS

You cannot use this function to cancel a send request that you made using the PNSendRequest function.

ASSEMBLY-LANGUAGE INFORMATION

To execute the PRelTCB function from assembly language, call the _Control trap macro with a value of relTCB in the csCode field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

noErr	0	No error
cbNotFound	–1102	The ATP control block was not found
noDataArea	–1104	Too many outstanding ATP functions

PKillGetReq

The `PKillGetReq` function cancels the pending `PGetRequest` function that you specify.

```
FUNCTION PKillGetReq (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`csCode`	`Integer`	Always `killGetReq` for this function.
→	`aKillQEl`	`Pointer`	A pointer to the queue element

Field descriptions

`aKillQEl` A pointer to the queue element of the pending call that is to be canceled.

DESCRIPTION

The `PKillGetReq` function lets you cancel a specific outstanding `PGetRequest` function without having to cancel all pending get requests or having to close the socket to do this; closing the socket cancels all outstanding functions on that socket.

To cancel a specific pending `PGetRequest` function, you specify the pointer to the queue element for the function in the `aKillQEl` field of the parameter block for the `PKillGetReq` function. The queue element pointer is the pointer to the parameter block of the `PGetRequest` function to be canceled. If the function has already completed execution or if it is not in the ATP queue for any other reason, `PKillGetReq` returns a message (`cbNotFound`) indicating that it could not find the parameter block.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PKillGetReq` function from assembly language, call the `_Control` trap macro with a value of `killGetReq` in the `csCode` field of the parameter block.
To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

`noErr`	0	No error
`cbNotFound`	–1102	The `aKilllQEl` parameter does not point to a `PGetRequest` queue element

ATPKillAllGetReq

The `ATPKillAllGetReq` function cancels all pending calls to the `PGetRequest` function for a specific socket.

```
FUNCTION ATPKillAllGetReq (thePBPtr: ATPPBPtr;
                           async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to the completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`csCode`	`Integer`	Always `killAllGetReq` for this function.
→	`atpSocket`	`Byte`	The socket number whose pending `PGetRequest` functions are to be canceled.

Field descriptions

`atpSocket` The socket whose pending `PGetRequest` functions are to be canceled.

DESCRIPTION

The `ATPKillAllGetReq` function cancels all pending `PGetRequest` functions issued on a specific socket without closing the socket. For each function executed asynchronously, `ATPKillAllGetReq` also calls the completion routine with the value `reqAborted` (–1105) in the D0 register. You should call the `ATPKillAllGetReq` function before closing a socket.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `ATPKillAllGetReq` function from assembly language, call the `_Control` trap macro with a value of `killAllGetReq` in the `csCode` field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

`noErr`	0	No error
`cbNotFound`	–1102	Control block not found; no pending asynchronous calls

PRelRspCB

The `PRelRspCB` function cancels a `PSendResponse` function that is an exactly-once transaction.

```
FUNCTION PRelRspCB (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

`thePBPtr` A pointer to an ATP parameter block.

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

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to the completion routine.
←	`ioResult`	`OSErr`	The function result.
→	`csCode`	`Integer`	Always `relRspCB` for this function.
→	`atpSocket`	`Byte`	The number of the socket on which the request was received.
→	`addrBlock`	`AddrBlock`	The internet socket address of the source of the request.
→	`transID`	`Byte`	The transaction ID of the request with which the `PSendResponse` function to be canceled is associated.

Field descriptions

`atpSocket` The number of the socket on which the request was received and from which the `PSendResponse` function that is to be canceled was sent.

`addrBlock` The internet socket address of the application that issued the request.

`transID` The transaction ID of the `PSendResponse` call to be canceled. You can get the transaction ID from the `reqTID` field of the `PSendResponse` parameter block queue entry.

DESCRIPTION

The `PRelRspCB` function releases the queued parameter block for the exactly-once transaction `PSendResponse` function without waiting for the release timer to expire or for a `TRel` packet to be received; `PRelRspCB` returns a function result of `noErr` for the canceled `PSendResponse` call.

If you call `PRelRspCB` to cancel a transaction that is not an exactly-once service, `RelRspCB` returns a function result of `cbNotFound` for the `PSendResponse` call.

ASSEMBLY-LANGUAGE INFORMATION

To execute the `PRelRspCB` function from assembly language, call the `_Control` trap macro with a value of `relRspCB` in the `csCode` field of the parameter block. To execute this function from assembly language, you must also specify the .ATP driver reference number.

RESULT CODES

`noErr`	0	No error
`cbNotFound`	–1102	Control block not found; no pending asynchronous calls

Building a Buffer Data Structure

You need to provide a buffer data structure (BDS) to hold data that comprises multiple response packets whether you are sending the response data or receiving it. This section describes a utility, `BuildBDS`, that ATP provides that allows you to create a BDS to be used for this purpose.

BuildBDS

From the buffer that you supply, the `BuildBDS` function creates a buffer data structure (BDS) to be used to hold data for ATP functions that send and receive response data.

```
FUNCTION BuildBDS (buffPtr: Ptr; bdsPtr: Ptr;
                   buffSize: Integer): Integer;
```

`buffPtr` A pointer to a data buffer.

`buffSize` The length in bytes of the buffer data structure.

DESCRIPTION

The `PSendResponse`, `PSendRequest`, and `PNSendRequest` functions require a buffer data structure of a specific format to be used to hold the response data. You can use the `BuildBDS` function to create this data structure, or you can build it yourself from Pascal.

The `BuildBDS` function creates a buffer data structure consisting of an array of elements—one for each response packet—to be used to hold response data. You pass this function a pointer to the memory to be used for this buffer and the size in bytes of the memory. You should allocate enough memory to hold the response data that you are either sending or receiving. Because an entire response message cannot exceed 4624 bytes, the amount of memory that you allocate for this data structure should not exceed this size.

`BuildBDS` creates up to eight elements for a buffer data structure. If you provide the maximum space of 4624 bytes, `BuildBDS` returns eight elements; if the response message is shorter and you specify fewer bytes, `BuildBDS` returns the equivalent number of elements. `BuildBDS` returns as a function result the number of buffer data structure elements that it creates. For more information about the BDS data structure, see "The Buffer Data Structure" on page 6-20.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Version number is too high

SEE ALSO

See "PSendResponse" on page 6-34, "PSendRequest" on page 6-24, and "PNSendRequest" on page 6-27 for more information about the functions that require a buffer data structure.

Summary of ATP

Pascal Summary

Constants

```
CONST
   {csCodes}
   nSendRequest         = 248;          {send request using a specific socket}
   relRspCB             = 249;          {release RspCB}
   closeATPSkt          = 250;          {close ATP socket}
   addResponse          = 251;          {add response}
   sendResponse         = 252;          {send response}
   getRequest           = 253;          {get request}
   openATPSkt           = 254;          {open ATP socket}
   sendRequest          = 255;          {send request}
   relTCB               = 256;          {release TCB}
   killGetReq           = 257;          {kill getRequest}
   killSendReq          = 258;          {kill sendRequest}
   killAllGetReq        = 259;          {kill all getRequests for a socket}

   {ATP flags}
   atpXOvalue              = 32;        {ATP exactly-once bit}
   atpEOMvalue             = 16;        {ATP end-of-message bit}
   atpSTSvalue             = 8;         {ATP send-transmission-status bit}
   atpTIDValidvalue        = 2;         {ATP trans. ID valid bit}
   atpSendChkvalue         = 1;         {ATP send checksum bit}
```

Data Types

The Buffer Data Structure

```
TYPE  BDSElement =
   RECORD
      buffSize:   Integer;
      buffPtr:    Ptr;
      dataSize:   Integer;
      userBytes:  LongInt;
   END;
```

```
BDSType = ARRAY[0..7] OF BDSElement;
BDSPtr = ^BDSType;
BitMapType = PACKED ARRAY[0..7] OF Boolean;
```

The Address Block Record

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

The ATP Parameter Block

```
TYPE ATPParamBlock =
   PACKED RECORD
      qLink:         QElemPtr;         {next queue entry}
      qType:         Integer;          {queue type}
      ioTrap:        Integer;          {routine trap}
      ioCmdAddr:     Ptr;              {routine address}
      ioCompletion:  ProcPtr;          {completion routine}
      ioResult:      OSErr;            {result code}
      userData:      Longint;          {ATP user bytes}
      reqTID:        Integer;          {request transaction ID}
      ioRefNum:      Integer;          {driver reference number}
      csCode:        Integer;          {call command code }
                                       { automatically set}
      atpSocket:     Byte;             {currBitMap or socket number}
      CASE MPPParmType OF
         SendRequestParm,
         SendResponseParm,
         GetRequestParm,
         AddResponseParm,
         KillSendReqParm:
            (atpFlags:        Byte;    {control information}
            addrBlock:        AddrBlock;
                                       {source/dest. socket address}
            reqLength:        Integer; {request/response length}
            reqPointer:       Ptr;     {ptr to request/response data}
            bdsPointer:       Ptr;     {ptr to response BDS}
         CASE MPPParmType OF
            SendRequestParm:
               (numOfBuffs:   Byte;    {number of responses expected}
```

```
            timeOutVal:     Byte;    {timeout interval}
            numOfResps:     Byte;    {number of responses }
                                     { actually received}
            retryCount:     Byte;    {number of retries}
            intBuff:        Integer; {used internally for PNSendRequest}
            TRelTime:       Byte);   {TRelease time for extended }
                                     { send request}
         SendResponseParm:
            (filler0:       Byte;    {numOfBuffs}
            bdsSize:        Byte;    {number of BDS elements}
            transID:        Integer);{transaction ID}
         GetRequestParm:
            (bitMap:        Byte;    {bitmap}
            filler1:        Byte);
         AddResponseParm:
            (rspNum:        Byte;    {sequence number}
            filler2:        Byte);
         KillSendReqParm:
            (aKillQEl:      Ptr));   {pointer to queue element to cancel}
   END;

ATPPBPtr = ^ATPParamBlock;
```

Routines

Sending an ATP Request

```
FUNCTION PSendRequest       (thePBPtr: ATPPBPt; async: Boolean): OSErr;
FUNCTION PNSendRequest      (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

Opening and Closing an ATP Socket

```
FUNCTION POpenATPSkt        (thePBptr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION PCloseATPSkt       (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

Setting Up a Socket to Listen for Requests

```
FUNCTION PGetRequest        (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

Responding to Requests

```
FUNCTION PSendResponse      (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION PAddResponse       (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

Canceling Pending ATP Functions

```
FUNCTION PKillSendReq       (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION PRelTCB            (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION PKillGetReq        (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION ATPKillAllGetReq   (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
FUNCTION PRelRspCB          (thePBPtr: ATPPBPtr; async: Boolean): OSErr;
```

Building a Buffer Data Structure

```
FUNCTION BuildBDS           (buffPtr: Ptr; bdsPtr: Ptr; buffSize: Integer):
                             Integer;
```

C Summary

Constants

```
/*ATP parameter constants*/
#define ATPioCompletion ATP.ioCompletion
#define ATPioResult ATP.ioResult
#define ATPuserData ATP.userData
#define ATPreqTID ATP.reqTID
#define ATPioRefNum ATP.ioRefNum
#define ATPcsCode ATP.csCode
#define ATPatpSocket ATP.atpSocket
#define ATPatpFlags ATP.atpFlags
#define ATPaddrBlock ATP.addrBlock
#define ATPreqLength ATP.reqLength
#define ATPreqPointer ATP.reqPointer
#define ATPbdsPointer ATP.bdsPointer
#define ATPtimeOutVal SREQ.timeOutVal
#define ATPnumOfResps SREQ.numOfResps
#define ATPretryCount SREQ.retryCount
#define ATPnumOfBuffs OTH1.u0.numOfBuffs
#define ATPbitMap OTH1.u0.bitMap
#define ATPrspNum OTH1.u0.rspNum
#define ATPbdsSize OTH2.bdsSize
#define ATPtransID OTH2.transID
#define ATPaKillQEl KILL.aKillQEl
```

```
/*csCodes*/
enum {                                      /*csCodes*/
   nSendRequest        = 248,               /*send request using a specific */
                                            /* socket*/
   relRspCB            = 249,               /*release RspCB*/
   closeATPSkt         = 250,               /*close ATP socket*/
   addResponse         = 251,               /*add response*/
   sendResponse        = 252,               /*send response*/
   getRequest          = 253,               /*get request*/
   openATPSkt          = 254,               /*open ATP socket*/
   sendRequest         = 255,               /*send request*/
   relTCB              = 256,               /*release TCB*/
   killGetReq          = 257,               /*kill getRequest*/
   killSendReq         = 258,               /*kill sendRequest*/
   killAllGetReq       = 259};              /*kill all getRequests for */
                                            /* a socket*/

/*ATP flags*/
enum {
   atpXOvalue             = 32,             /*ATP exactly-once bit*/
   atpEOMvalue            = 16,             /*ATP end-of-message bit*/
   atpSTSvalue            = 8,              /*ATP send-transmission-status */
                                            /* bit*/
   atpTIDValidvalue       = 2,              /*ATP trans. ID valid bit*/
   atpSendChkvalue        = 1};             /*ATP send checksum bit*/
```

Data Types

The Buffer Data Structure

```
struct   BDSElement {
   short       buffSize;
   Ptr         buffPtr;
   short       dataSize;
   long        userBytes;
};

typedef struct BDSElement BDSElement;

typedef BDSElement BDSType[8];
typedef BDSElement *BDSPtr;
typedef char BitMapType;
```

The Address Block Record

```
struct AddrBlock {
   short            aNet;
   unsigned char    aNode;
   unsigned char    aSocket;
};

typedef struct AddrBlock AddrBlock;
```

The ATP Parameter Block

```
#define MPPATPHeader \
   QElem       *qLink;                 /*next queue entry*/\
   short       qType;                  /*queue type*/\
   short       ioTrap;                 /*routine trap*/\
   Ptr         ioCmdAddr;              /*routine address*/\
   ProcPtr     ioCompletion;           /*completion routine*/\
   OSErr       ioResult;               /*result code*/\
   long        userData;               /*command result (ATP user bytes)*/\
   short       reqTID;                 /*request transaction ID*/\
   short       ioRefNum;               /*driver reference number*/\
   short       csCode;                 /*call command code*/

typedef struct {
   MPPATPHeader
}MPPparms;

#define MOREATPHeader \
   char        atpSocket;              /*currbitmap for requests or ATP */\
                                       /* socket number*/\
   char        atpFlags;               /*control information*/\
   AddrBlock   addrBlock;              /*source/dest. socket address*/\
   short       reqLength;              /*request/response length*/\
   Ptr         reqPointer;             /*pointer to request/response data*/\
   Ptr         bdsPointer;             /*pointer to response BDS*/

typedef struct {
   MPPATPHeader
   MOREATPHeader
}ATPparms;
```

```
typedef struct {
   MPPATPHeader
   MOREATPHeader
   char          filler;              /*numOfBuffs*/
   char          timeOutVal;          /*timeout interval*/
   char          numOfResps;          /*number of responses actually */
                                      /* received*/
   char          retryCount;          /*number of retries*/
   short         intBuff;             /*used internally for NSendRequest*/
   char          TRelTime;            /*TRelease time for extended send */
                                      /* request*/
}SendReqparms;

typedef struct {
    MPPATPHeader
    MOREATPHeader
   union {
      char     bitMap;                /*bitmap received*/
      char     numOfBuffs;            /*number of responses being sent*/
      char     rspNum;                /*sequence number*/
 } u0;
}ATPmisc1;

typedef struct {
   MPPATPHeader
   MOREATPHeader
   char            filler;
   char            bdsSize;           /*number of BDS elements*/
   short           transID;           /*transaction ID*/
}ATPmisc2;

typedef struct {
   MPPATPHeader
   MOREATPHeader
   Ptr             aKillQEl;          /*pointer to i/o queue element to */
                                      /* cancel*/
}Killparms;

union ATPParamBlock {
   ATPparms        ATP;               /*general ATP parms*/
   SendReqparms    SREQ;              /*send request parms*/
   ATPmisc1        OTH1;              /*miscellaneous parms*/
   ATPmisc2        OTH2;              /*miscellaneous parms*/
   Killparms       KILL;              /*kill request parms*/
};
```

```
typedef union ATPParamBlock ATPParamBlock;
typedef ATPParamBlock *ATPPBPtr;
```

Routines

Sending an ATP Request

```
pascal OSErr PSendRequest    (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr PNSendRequest   (ATPPBPtr thePBPtr,Boolean async);
```

Opening and Closing an ATP Socket

```
pascal OSErr POpenATPSkt     (ATPPBPtr thePBptr,Boolean async);
pascal OSErr PCloseATPSkt    (ATPPBPtr thePBPtr,Boolean async);
```

Setting Up a Socket to Listen for Requests

```
pascal OSErr PGetRequest     (ATPPBPtr thePBPtr,Boolean async);
```

Responding to Requests

```
pascal OSErr PSendResponse   (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr PAddResponse    (ATPPBPtr thePBPtr,Boolean async);
```

Canceling Pending ATP Functions

```
pascal OSErr PKillSendReq    (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr PRelTCB         (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr PKillGetReq     (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr ATPKillAllGetReq
                             (ATPPBPtr thePBPtr,Boolean async);
pascal OSErr PRelRspCB       (ATPPBPtr thePBPtr,Boolean async);
```

Building a Buffer Data Structure

```
pascal short BuildBDS        (Ptr buffPtr,Ptr bdsPtr,short buffSize);
```

Assembly-Language Summary

Constants

ATP Header

```
atpControl      EQU   0           ;control field (byte)
atpBitmap       EQU   1           ;bitmap (requests only) (byte)
atpRespNo       EQU   1           ;response number (responses only) (byte)
atpTransID      EQU   2           ;transaction ID (word)
atpUserData     EQU   4           ;start of user data (long)
atpHdSz         EQU   8           ;size of ATP header
```

ATP Control Field

```
atpReqCode      EQU   $40         ;request code after masking
atpRspCode      EQU   $80         ;response code after masking
atpRelCode      EQU   $C0         ;release code after masking
atpXOBit        EQU   5           ;bit number of exactly-once bit
atpEOMBit       EQU   4           ;bit number of end-of-message bit
atpSTSBit       EQU   3           ;send transmission status bit number
flagMask        EQU   $3F         ;mask for just flags
controlMask     EQU   $F8         ;mask for good control bits
```

ATP Type Code

```
atp             EQU   $3          ;ATP type code (in DDP header)
```

ATP Limits

```
atpMaxNum       EQU   8           ;maximum number of responses per request
atpMaxData      EQU   $242        ;maximum data size in ATP packet
```

ATP Command Codes

```
nSendRequest    EQU   248         ;PNSendRequest code
relRspCB        EQU   249         ;release RspCB
closeATPSkt     EQU   250         ;close ATP socket
addResponse     EQU   251         ;add response code
sendResponse    EQU   252         ;send response code
getRequest      EQU   253         ;get request code
```

```
openATPSkt      EQU   254         ;open ATP socket
sendRequest     EQU   255         ;send request code
relTCB          EQU   256         ;release TCB
killGetReq      EQU   257         ;kill GetRequest
killSendReq     EQU   258         ;kill SendRequest
killAllGetReq   EQU   259         ;kill all getRequests for a socket
```

ATPQueue Element Standard Structure

```
;arguments passed in the CSParam area

atpSocket       EQU   $1C         ;socket number is first parameter [byte]
atpFlags        EQU   $1D         ;flag [byte]
addrBlock       EQU   $1E         ;start of address block
reqLength       EQU   $22         ;size of request buffer [word]
reqPointer      EQU   $24         ;pointer to request buffer or data
bdsPointer      EQU   $28         ;pointer to buffer data structure (BDS)
guArea          EQU   $2C         ;start of general-use area
userData        EQU   $12         ;user bytes
```

ATP Bits

```
sendCHK         EQU   0           ;bit number of send-checksum bit in flags
tidValid        EQU   1           ;bit set when TID valid in SendRequest
```

Data Structures

Buffer Data Structure (BDS)

```
bdsBuffSz       EQU   0           ;send: data length
                                  ; receive: buffer length
bdsBuffAdr      EQU   2           ;send: data address
                                  ; receive: buffer address
bdsDataSz       EQU   6           ;send: used internally
                                  ; receive: data length
bdsUserData     EQU   8           ;send: 4 user bytes
                                  ; receive: 4 user bytes
bdsEntrySz      EQU   12          ;size of a BDS entry
```

ATP Parameter Block Common Fields

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	userData	long	user bytes
22	reqTID	word	request transaction ID
24	ioRefNum	word	driver reference number
26	csCode	word	command code
28	atpSocket	byte	current bitmap or socket number

SendRequest Parameter Variant

26	csCode	word	command code; always sendRequest
28	currBitMap	byte	current bitmap
29	atpFlags	byte	control information
30	addrBlock	long	destination socket address
34	reqLength	word	request size in bytes
36	reqPointer	long	pointer to request data
40	bdsPointer	long	pointer to response BDS
44	numOfBuffs	byte	number of responses expected
45	timeOutVal	byte	timeout interval
46	numOfResps	byte	number of responses received
47	retryCount	byte	number of retries
50	TrelTime	byte	release time for extended send request

NSendRequest Parameter Variant

22	reqTID	word	request transaction ID
26	csCode	word	command code; always nSendRequest
29	atpFlags	byte	control information
30	addrBlock	long	destination socket address
34	reqLength	word	request size in bytes
36	reqPointer	long	pointer to request data
40	bdsPointer	long	pointer to response BDS
44	numOfBuffs	byte	number of responses expected
45	timeOutVal	byte	timeout interval
46	numOfResps	byte	number of responses received
47	retryCount	byte	number of retries
50	TrelTime	byte	release time for extended send request

OpenATPSkt Parameter Variant

26	csCode	word	command code; always openATPSkt
30	addrBlock	long	socket request specification

CloseATPSkt Parameter Variant

26	`csCode`	word	command code; always `closeATPSkt`

GetRequest Parameter Variant

22	`reqTID`	word	request transaction ID
26	`csCode`	word	command code; always `getRequest`
29	`atpFlags`	byte	control information
30	`addrBlock`	long	destination socket address
34	`reqLength`	word	request size in bytes
36	`reqPointer`	long	pointer to request data
44	`bitMap`	byte	current bitmap

SendResponse Parameter Variant

26	`csCode`	word	command code; always `sendResponse`
29	`atpFlags`	byte	control information
30	`addrBlock`	long	destination socket address
40	`bdsPointer`	long	pointer to response BDS
44	`numOfBuffs`	byte	number of responses expected
45	`bdsSize`	byte	BDS size in elements
46	`transID`	word	transaction ID

AddResponse Parameter Variant

26	`csCode`	word	command code; always `addResponse`
29	`atpFlags`	byte	control information
30	`addrBlock`	long	destination socket address
34	`reqLength`	word	response size in bytes
36	`reqPointer`	long	pointer to response data
44	`rspNum`	byte	sequence number
46	`transID`	word	transaction ID

KillSendReq Parameter Variant

26	`csCode`	word	command code; always `killSendReq`
44	`aKillQEl`	long	pointer to queue element of function to be removed

RelTCB Parameter Variant

26	`csCode`	word	command code; always `relTCB`
30	`addrBlock`	long	destination socket address of request
46	`transID`	word	transaction ID of request to be canceled

KillGetReq Parameter Variant

26	`csCode`	word	command code; always `killGetReq`
44	`aKillQEl`	long	pointer to queue element of function to be removed

KillAllGetReq Parameter Variant

26	csCode	word	command code; always killAllGetReq

RelRspCB Parameter Variant

26	csCode	word	command code; always relRspCB
30	addrBlock	long	internet socket address of the source of the request
46	transID	word	transaction ID of request with which the PSendResponse function to be canceled is associated

Result Codes

noErr	0	No error
paramErr	–50	Version number is too high
reqFailed	–1096	Retry count exceeded
tooManyReqs	–1097	Too many concurrent requests
tooManySkts	–1098	Too many responding sockets
badATPSkt	–1099	Bad responding socket
badBuffNum	–1100	Sequence number out of range
noRelErr	–1101	No release received
cbNotFound	–1102	The aKillQEl parameter does not point to a PSendRequest or PNSendRequest queue element
noSendResp	–1103	PAddResponse issued before PSendResponse
noDataArea	–1104	Too many outstanding ATP calls
reqAborted	–1105	Request canceled