SONY

OPEN-R SDK

OPEN-R Internet Protocol Version4

115-01

Notes on This Document

Notes on Using This Document

- The contents provided by this document (PDF files) are intended only for supplying informaion.
- The contents provided by this document (PDF files) are subject to change without notice.
- We are not responsible for errors or omissions in technical or editorial aspects concerning the contents described in this document. We also are not responsible for technical measures, correspondence, execution according to this document, as well as for the results occurred by them such as inevitable, indirect or incidental damages.

Notes on Copyright

About Trademarks

Index

About this book

This book describes the Internet Protocol version 4 (IPv4) implemented on OPEN-R working on OPEN-R1.1.3. This release of IPv4 includes four network protocols: TCP, UDP, IP and the client side of DHCP.

This book has two parts:

- IPv4 Programmer's Guide – An introduction to networking on OPEN-R and to the IPv4 protocol stack. Detailed instructions are given for how OPEN-R objects can use the TCP, UDP, DNS, and IP services in the stack.

- IPv4 Reference – Detailed descriptions of all classes, messages, errors, and operations that you will encounter when writing OPEN-R objects that communicate with the IPv4 protocol stack.

OPEN-R objects are described as "objects" in this manual. In case a remote side is not an OPEN-R host(e.g. UNIX), it is more appropriate to call it a process instead of object, but we use "object" as a generic term in this manual.

Part1 IPv4 Programmer's Guide

1 Introduction to the IPv4 protocol stack

This chapter describes the protocols in the current version of the IPv4 protocol stack, introduces this implementation of that IPStack on OPEN-R, and explains how objects communicate with the stack.

1.1 Protocols in the IPv4 protocol stack

The Internet Protocol (IP) is a protocol for sending data between hosts on the Internet. IP version 4 (IPv4) is currently the most widely used version of this protocol, and is the version available on OPEN-R. The IPv4 protocol stack on OPEN-R includes several protocols that supplement the basic IP protocol.

This release of the IPv4 protocol stack contains the following protocols:

- IP (Internet Protocol) – The base protocol, responsible for delivering datagrams over the Internet. This is a packet-oriented, connectionless protocol, offering unreliable transfer of IP datagrams.
- TCP (Transmission Control Protocol) – Runs on top of the IP protocol. It provides objects with a connection-oriented, reliable, byte stream service.
- UDP (User Datagram Protocol) – Runs on top of the IP protocol. It provides objects with an unreliable datagram delivery service.
- DNS (Domain Name System) – A service for mapping domain names to IP addresses and vice-versa.
- DHCP (Dynamic Host Configuration Protocol) – A service for allocating reusable network addresses and additional configuration options.

1.2 The IPv4 protocol stack

On OPEN-R, the IPv4 protocol stack is implemented using the OPEN-R Networking Toolkit (ANT). The stack exists at runtime in the IPStack, which also includes the ANT runtime environment. The IPStack is an OPEN-R system layer object.

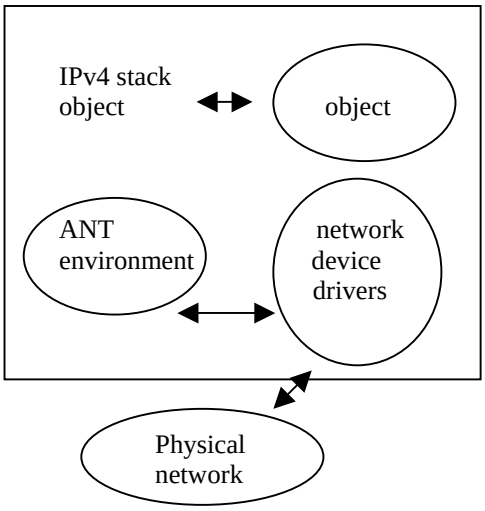

Figure 1 The IPStack provides networking services on OPEN-R.

The IPStack communicates through message passing with objects and with device drivers.

1.3 How your object communicates with the protocol stack

Objects can use the network services offered by the IPv4 protocol stack. Objects communicate with the protocol stack through normal message passing, by sending special messages to and receiving special messages from the IPStack.

The first thing that an object must do is ask the IPStack to create an endpoint. An endpoint is a special ANT construct that is located at the top of the protocol stack and is responsible for communication between the object and the stack. Typically, an object requires one endpoint per network connection. For example, an object that needs to send data through a TCP connection and a UDP connection would require two endpoints. So would an object with two TCP connections. Instructions on how to create new endpoints are described later.

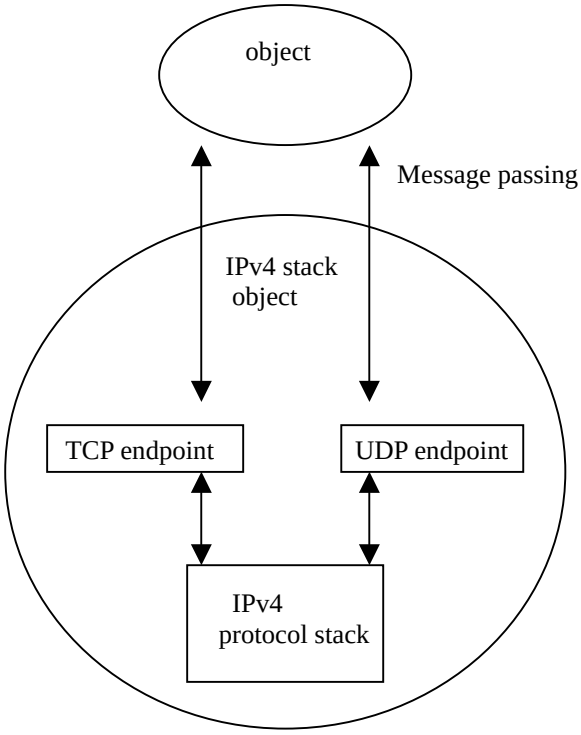

Figure 2 Objects send requests for network services to endpoints in the IPStack. These endpoints provide access to the IPv4 protocol stack.

In addition to its endpoints, an object must also create one or more shared memory buffers. Shared memory buffers are required for exchanging data between objects and the IPv4 protocol stack. Objects and the protocol stack do not necessarily share the same address space, so they cannot exchange pointers to the data being transferred. Shared memory buffers, implemented by the antSharedBuffer structure, map a common memory area into the address space of both objects. Instructions on how to create shared memory buffers are described later.

All messages sent to the IPStack are inherited from the antEnvMsg structure. This structure provides the basic message handling constructs such as Send() and Call(). Each service offered by a protocol has a specific inherited message type: for example, a request to send data by TCP is made with a TCPEndPointSendMsg, and a request to bind a UDP connection is made with a UDPEndPointBindMsg.

Creating new endpoints

To create a new endpoint for any protocol in the IP stack, your object sends an antEnvCreateEndpointMsg to the IPStack. In the antEnvCreateEndpointMsg, your object specifies which protocol the endpoint is required to implement, and how much memory should be allocated to the endpoint's SDU pool. The IPStack creates a new endpoint for the specified protocol and replies to the antEnvCreateEndpointMsg. The ModuleRef parameter in this message now stores a reference to the new endpoint.

Example
In this example, an object creates a new TCP endpoint. This process is exactly the same for other protocols in the IPStack, except the object would specify a different type of endpoint. See the $IPv4_ROOT/Examples/TCP directory for the complete TCP example.

To begin, the object creates a message that requests a new TCP endpoint:

```
antEnvCreateEndpointMsg createMsg(
  EndpointType_TCP,
  4 * PACKETSIZE
);
```

The following endpoint types are available:

- EndpointType_TCP
- EndpointType_UDP
- EndpointType_DNS
- EndpointType_IP

Notice that the object requests an SDU pool that is four times greater than the packet size (4 * PACKETSIZE). An SDU pool is an internal ANT construct that stores data in the protocol stack. As a guideline, always create an SDU pool that is slightly larger than the largest packet that you expect to send. For example, 8-KB packets would require an SDU pool of approximately 10 KB.

The object now sends the message to the IPStack:

```
createMsg.Call(
  IPStackRef,
  sizeof(antEnvCreateEndpointMsg)
);
```

The Call() method is inherited from antEnvMsg. It specifies an antStackRef to the IPStack (IPStackRef), and the size of the message. The Call() method sends the message synchronously, which means that the object will continue only when it receives a reply.

When the object receives a reply, it gets a reference to the new endpoint:

```
endpoint = createMsg.moduleRef;
```

This reference enables the object to communicate with the new endpoint. For the TCP protocol, the object can send messages that request services such as connecting, listening, sending, receiving, and closing. Other protocols may offer different services.

Creating shared memory buffers

Shared memory buffers are implemented by the antSharedBuffer structure. They map a shared memory area to the address spaces of your object and the protocol stack. When your object exchanges data with the protocol stack, the data is identified by a pointer to the shared buffer and an offset in the buffer. antSharedBuffer automatically converts this offset to a location in the object's address space, as shown in Figure 3.

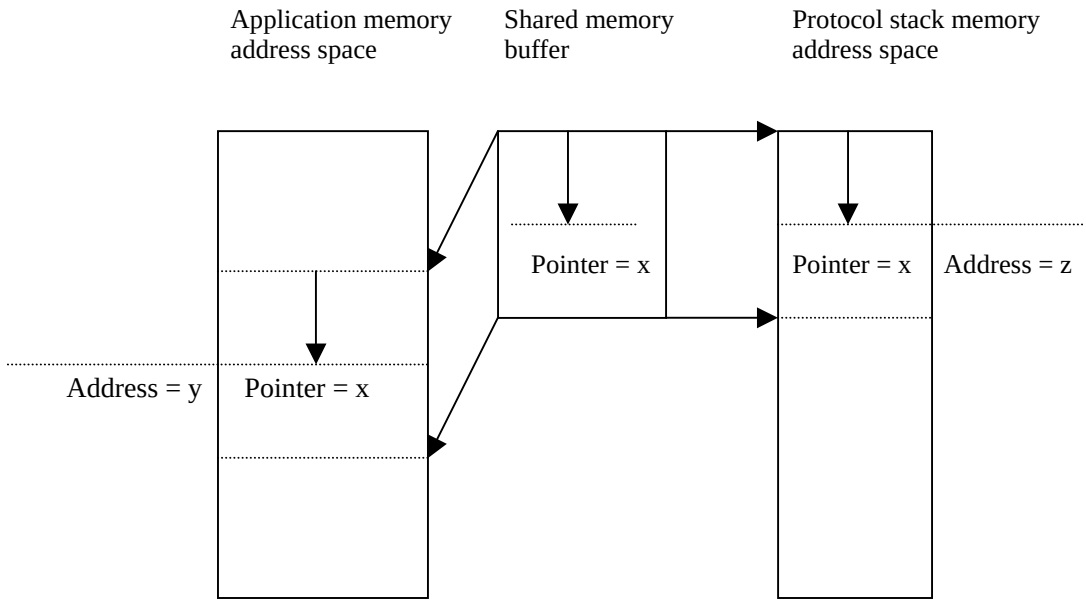

Figure 3 Shared memory buffers map memory areas to the address spaces of objects and the protocol stack, enabling them to exchange pointers to data.

Your object can create a single shared buffer, or it can create several buffers for specific operations such as send and receive.
Shared buffers are allocated and deallocated by the OPEN-R shared memory management routines. These routines can take a long time to execute, so you should create and delete shared buffers only when necessary. Try to reuse existing buffers whenever possible. Typically, you would create a send buffer and a receive buffer for every new endpoint, and then reuse these buffers for all subsequent send and receive operations on that endpoint.

There are also some restrictions on the size of shared buffers. Normally, the size that you request will be rounded up to the nearest multiple of the page size on OPEN-R. The page size of OPEN-R in memory protection mode is 4096 bytes. Therefore, using this mode, if you request a 7000-byte shared buffer, the buffer will be 8192 bytes (4096*2).

You can use shared buffers in many ways, but the needs of your particular object will help you to decide which approach to follow.

To create a shared buffer, your object sends an antEnvCreateSharedBufferMsg to the IPStack, specifying the size of the buffer. The IPStack creates the shared buffer and replies to the antEnvCreateSharedBufferMsg. The buffer parameter in this message now stores a reference to the new antSharedBuffer. Your object then maps the buffer to its address space, by invoking the antSharedBuffer::Map() method.

For descriptions of antEnvCreateSharedBufferMsg and antSharedBuffer, see the "ANT environment reference".

Example
In this example, a TCP client object creates two shared buffers: one for data being sent, and one for data being received.
To begin, the object creates a message that requests a new shared buffer:

```
antEnvCreateSharedBufferMsg bufferMsg(PACKETSIZE);
```

The specified buffer size is the same as the packet size, because the object will exchange data with the protocol stack only one packet at a time. As a general guideline, a shared memory buffer should be able to hold the largest packet that will be sent or received by the object.

The object now sends the message to the IPStack:

```
bufferMsg.Call(
  IPStackRef,
  sizeof(antEnvCreateSharedBufferMsg)
);
```

The Call() method is inherited from antEnvMsg. It specifies an antStackRef to the IPStack (IPStackRef), and the size of the message. The Call() method sends the message synchronously, which means the object will continue only when it receives a reply.

When the object receives a reply, it gets a reference to the new shared buffer and defines it as a send buffer:

```
sendBuffer = bufferMsg.buffer;
```

Finally, the object maps the shared buffer to its address space:

```
sendBuffer.Map()
```

To create a receive buffer, the object repeats the above process by once again sending the antEnvCreateSharedBufferMsg to the IPStack:

```
bufferMsg.Call(
  IPStackRef,
  sizeof(antEnvCreateSharedBufferMsg)
);

receiveBuffer = bufferMsg.buffer;
receiveBuffer.Map()
```

When the object sends data to the protocol stack, it sends a pointer to the data inside the send buffer. When it receives data from the protocol stack, it will receive a pointer to data in the receive buffer. The antSharedBuffer structure automatically converts these pointers to locations in the address space of the object or protocol stack.

Requesting network services

To request a service from a protocol in the IP protocol stack, an object creates a message and sends it to the appropriate endpoint in the IPStack. The message type identifies which service is required, and the message contents include the information needed to perform the service (such as IP addresses, port numbers, pointers to data in shared buffers, and so on).

Each protocol offers a unique set of messages that have been inherited from antEnvMsg. For descriptions of these messages, see the TCP, UDP, DNS, and IP sections of the IPv4 Reference.

Example
In this example, an object opens a TCP connection to another host.
To begin, it creates a message that requests the connection:

```
TCPEndpointConnectMsg connectMsg(
  endpoint,
  0,
  0,
  "193.74.243.95",
  7
);
```

The first parameter, endpoint, identifies the TCP endpoint to which the message will be sent. The next two parameters, both 0, will return the local IP address and port number when the connection is established. The last two parameters specify the IP address and port number of the host to which the connection should be established.

The object now sends the message to the TCP endpoint in the IPStack:

```
connectMsg.Call(
  IPStackRef,
  sizeof(TCPEndpointConnectMsg)
);
```

The Call() method is inherited from antEnvMsg. It specifies an antStackRef to the IPStack (IPStackRef), and the size of the message. The Call() method sends the message synchronously, which means the object will continue only when it receives a reply.

2 TCP guide

This chapter introduces the TCP protocol on OPEN-R, and explains how your object can use the TCP services offered by the IPv4 protocol stack.

2.1 TCP

TCP (Transmission Control Protocol) runs on top of the IP layer in the IPv4 protocol stack. TCP provides objects with a reliable, byte stream service. TCP is a connection-oriented protocol, so the sending and receiving objects need to establish a connection before any data transfer can take place.

TCP network operations

On OPEN-R, the IPv4 protocol stack offers the following TCP operations to objects:

- Connect – Open a TCP connection to an object on another host.
- Listen – Start listening for connection requests. This operation is typically performed by server objects instead of a connect operation. Server objects normally accept incoming TCP connections from client objects.
- Send – Send data over an open connection.
- Receive – Receive data from an open connection.
- Close – Close a TCP connection.

Your object performs these operations by sending special messages to a TCP endpoint in the IPStack. These messages are inherited from TCPEndpointBaseMessage, which is itself inherited from antEnvMsg. For descriptions of these messages, see “Chapter7 TCP reference.”

For an overview of how objects create endpoints and request network services, see “1.3 How your object communicates with the protocol stack.”

TCP endpoint life cycle

Figure 4 shows the state transitions of a TCP endpoint during its life cycle. These transitions illustrate the possible sequence of events and operations in a TCP connection. The message types shown in Figure 4 are described fully in the IPv4 Reference.

Your object requires one endpoint for each open network connection. You cannot request a new connection if a connection has already been established—your object must create a new endpoint first.

An endpoint can perform only one similar operation at a time. For example, if you send a TCPEndpointListenMsg to an endpoint that is already listening, the endpoint will return a TCP_CONNECTION_BUSY or TCP_OPERATION_INVALID error. However, it is possible to perform a send and receive operation at the same time.

2.2 Creating a TCP endpoint

Before your object can open a TCP connection, it must create a new TCP endpoint. Your object requires a new endpoint for each TCP connection that it opens. The process for creating an endpoint is the same for each protocol in the IPStack, and is described in detail in “How your object communicates with the protocol stack.”

See the /$IPv4_ROOT/Examples/TCP directory for a full working TCP example.

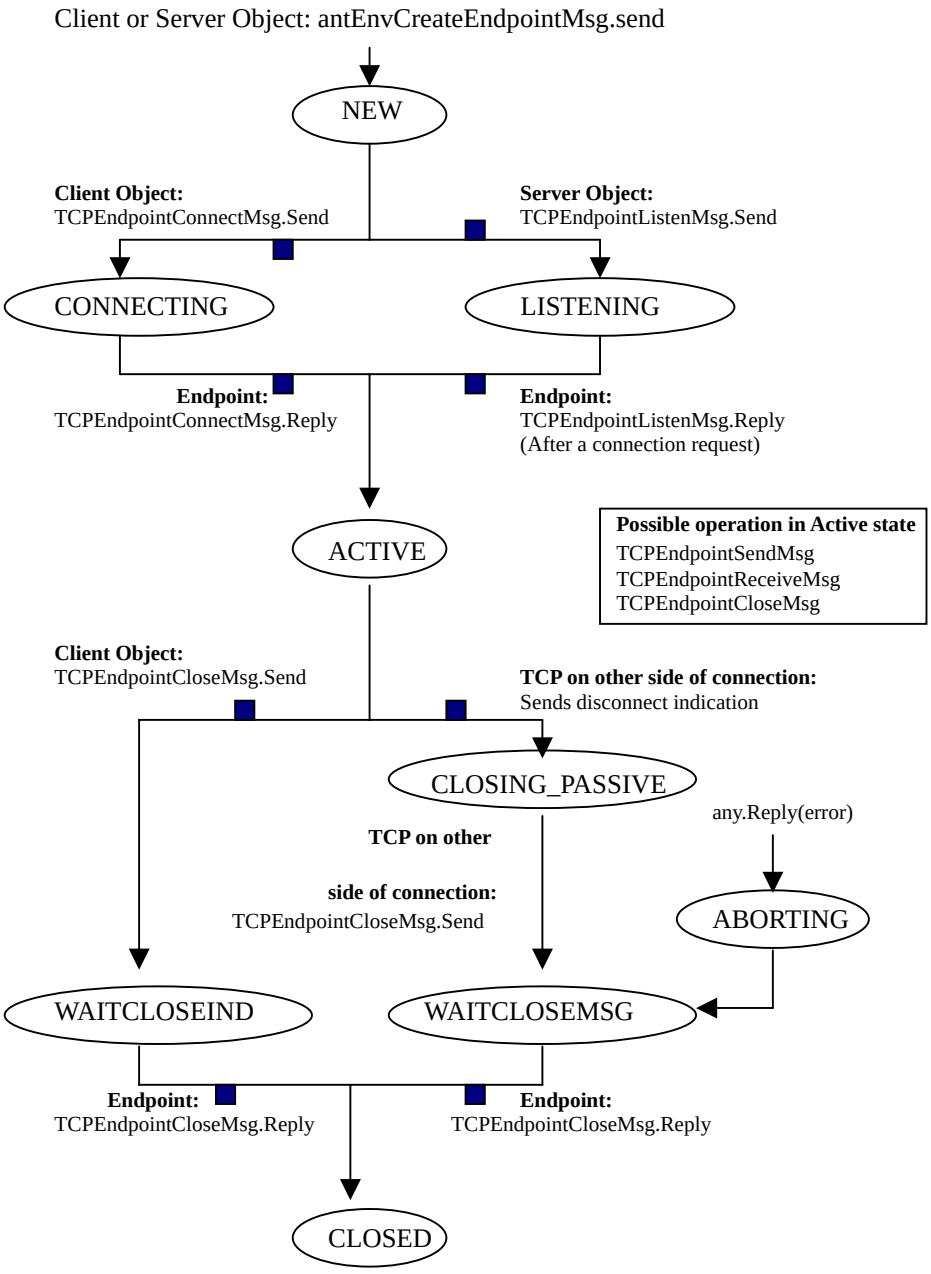

Figure 4 The state transitions of a TCP endpoint

2.3 Establishing a connection (client side)

When a client object needs to establish a TCP connection with a server, it sends a TCPEndpointConnectMsg to its endpoint in the IPStack. Table 1 shows the parameters in this message.

Table 1 Parameters in TCPEndpointConnectMsg

Parameter	Description
IPAddresslAddress	(out) Returns the local IP address, when the connection has been established.
PortlPort	(out) Returns an ephemeral port number assigned to the client object, when the connection has been established.
IPAddressfAddress	(in) The IP address of the host that you need to connect to.
PortfPort	(in) The port number of the object that you need to connect to.

When a connection is established, the TCP endpoint returns the fully specified IP addresses and port numbers. The objects on the two hosts can now send and receive data over the connection.

2.4 Listening for connection requests (server side)

A server object accepts connection requests from client objects. This means that the server object must listen for requests, and establish a connection only when it receives a request. Normally, it will accept requests only for connections to a specific port number. For example, an FTP server accepts connections only on the FTP port, equal to 21.

For a server object to handle multiple clients concurrently, it must perform more than one listen operation. Each listen operation waits for connection requests on the same port number. The server object requires a separate endpoint for each listen operation.

To start listening for connection requests, your server object sends a TCPEndpointListenMsg to its endpoint in the IPStack. Table 2 shows the parameters in this message.

Table 2 Parameters in TCPEndpointListenMsg

Parameter	Description
IPAddresslAddress	(out) Returns the local IP address when a connection has been established.
PortlPort	(in) The port number for which you will accept connection requests. If you will accept requests for any port, specify a value of IP_PORT_ANY.
IPAddressfAddress	(out) Returns the IP address of the host that requested the connection.
PortfPort	(out) Returns the port number of the object that requested the connection.

When a connection request is received, the TCP endpoint establishes the connection and returns the fully specified IP addresses and port numbers.

2.5 Sending data

To send data over an open TCP connection, your object sends a TCPEndpointSendMsg to its endpoint in the IPStack. Table 3 shows the parameters in this message.

Table 3 Parameters in TCPEndpointSendMsg

Parameter	Description
buffer	(in) Pointer to the data being sent. The data must be stored in a shared memory buffer defined by the antSharedBuffer structure.
size	(in) The size, in bytes, of the data being sent.

The TCP endpoint replies to this message when the data has been processed by the IP stack and the buffer can be reused.

2.6 Receiving data

To receive data from an open TCP connection, your object sends a TCPEndpointReceiveMsg to its TCP endpoint in the IPStack. Table 4 shows the parameters in this message.

Table 4 Parameters in TCPEndpointReceiveMsg

Parameter	Description
buffer	(in) Pointer to a memory area where the incoming data should be written. This area must be in a shared memory buffer defined by the antSharedBuffer structure.
sizeMin	(in/out) Specifies the minimum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received.
sizeMax	(in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received.

The TCP endpoint replies to this message when the data has been copied into the receive buffer. Note that when all data in the transmission has been received and the TCP connection is closed, the last receive request may hold a smaller number of bytes than what is specified in sizeMin.

2.7 Closing a connection

A TCP connection can be closed three different ways:

- Active close – The close request is sent directly by your object.
- Passive close – The close request is sent by the object on the other side of the connection.
- Abort – An error occurs, which closes the connection unexpectedly.

The following sections provide details on each of these methods of closing a connection.

Active close

An active close occurs when your object initiates the closing of the TCP connection. To perform an active close, your object sends a TCPEndpointCloseMsg to its endpoint in the IPStack.

When your object performs an active close, it can no longer send or receive data. The object on the other side of the connection will receive the rest of the transmission, and then it will receive an indication that the connection has been closed. From the perspective of the other object, a passive close has occurred.

Passive close

A passive close occurs when the object on the other side of the network closes the TCP connection (by performing an active close). After your object has received the entire data transmission, a TCP_CONNECTION_CLOSED error will occur. Your object must then complete the passive close by sending a TCPEndpointCloseMsg to its endpoint in the IPStack.

Abort

An abort occurs when something unexpected happens to the TCP connection. For example:

- The connection has been lost and it times out.
- A connection request takes too much time and is aborted by the requesting object.
- A normal active or passive close takes too much time and is aborted by one of the objects.
- An object decides to abort the connection.

An abort purges all data from the shared buffers and immediately closes the connection.

To abort a connection, your object sends a TCPEndpointCloseMsg to its endpoint in the IPStack. This message has one parameter, a boolean called abort. If TRUE, the TCP connection is aborted, instead of being shut down in an orderly fashion.

2.8 TCP echo client example

This section provides an example of a TCP echo client program, which illustrates how to use the TCP messages.

The TCP echo client establishes a connection to a TCP echo server, transfers some data, and closes the connection when it receives its data back from the server.

In this example, it should be possible to send and receive data at the same time. Therefore, the asynchronous version of antEnvMsg method is used in certain cases. This means that the number of entrypoints and the stubs to process these messages must be defined. See “Entry point definition” for numbering the entry points.

Variable definitions

```
#include <ant.h>
#include <EndpointTypes.h>
#include <TCPEndpointMsg.h>

The entrypoints:
enum
{
  Entry_Initialize = 0,
  Entry_ReceiveCont = 1,
  Entry_SendCont = 2,
  Echo_NumEntries = 3
};
OPEN-R OID and antStackRef are necessary for the message
passing:
OID myOID;
```

```
antStackRef IPStackRef;

Predefined values used by the client:
#define ECHOSERVER_IP "168.1.2.3"
#define ECHOSERVER_PORT 7

#define PACKETSIZE 1024
#define ECHOSIZE PACKETSIZE * 100
#define POOLSIZE 4096

The shared buffers used to transfer data:
antSharedBuffer sendBuffer;
byte* sendData;
antSharedBuffer receiveBuffer;
byte* receiveData;

The TCP connection:
antModuleRef connection;
uint32 bytesSent;
uint32 bytesReceived;
```

Entry point definition

Entry points are defined for the following processes.

- Object initialization
- Send request has been processed
- Receive request has been completed

The following description is needed for stub.cfg. See “Extra entry, 2.3 Stub in Programmer’s Guide” for the details of the definition of entry points.

```
Extra: Initialize()
Extra: ReceiveCont()
Extra: SendCont()
```

Executing a stubgen2 command generates xxxStub.cc where the following are described.

```
            :
GEN_ENTRY(_Initializestub,      _Initialize);
GEN_ENTRY(_ReceiveContstub,     _ReceiveCont);
GEN_ENTRY(_SendContstub,        _SendCont);

ObjectEntry ObjectEntryTable[]= {
            :
     {Extra_Entry[0],    (Entry)_Initializestub},
     {Extra_Entry[1],    (Entry)_ReceiveCont},
     {Extra_Entry[2],    (Entry)_SendCont},
     {UNIDEF,            (Entry)_ENTRY_UNDEF}
};
```

The number of Entry points is determined by the order of the elements in ObjectEntryTable[].

Initialize the object

This function is invoked when the client object starts up. It performs the following operations:

- Create the shared buffers
- Create the TCP endpoint
- Connect to the TCP echo server

```
void Initialize(void)
{
Get some OIDs for the OPEN-R message passing:
  WhoAmI(&myOID);
  IPStackRef = antStackRef("IPStack");
Initialize counters:
  bytesSent = 0;
  bytesReceived = 0;

Allocate a shared buffer for receiving data:
  antEnvCreateSharedBufferMsg receiveBufferMsg(PACKETSIZE);
  receiveBufferMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateSharedBufferMsg)
  );

  if (ANT_SUCCESS != receiveBufferMsg.error)
  {
    EXIT();
  }
  receiveBuffer = receiveBufferMsg.buffer;
  receiveBuffer.Map();

Store the base address of the receive buffer:
  receiveData = (byte*)receiveBuffer.GetAddress();
Do the same steps to get the send buffer:
  sendData = (byte*)sendBuffer.GetAddress();
Create a TCP connection:
  antEnvCreateEndpointMsg           createMsg(EndpointType_TCP,
POOLSIZE);
  createMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateEndpointMsg)
  );

  if (ANT_SUCCESS != createMsg.error)
  {
    EXIT();
  }
  connection = createMsg.moduleRef;
Connect to the echo server:
  TCPEndpointConnectMsg connectMsg(
    connection,
    IP_ADDR_ANY,
    IP_PORT_ANY,
    ECHOSERVER_IP,
    ECHOSERVER_PORT
  );
  connectMsg.Call(
    IPStackRef,
    sizeof(TCPEndpointConnectMsg)
  );
  if (TCP_SUCCESS != connectMsg.error)
  {
```

```
    EXIT()
  }
The connection is now established. To get information about
the     connection,     you     examine     connectMsg.lAddress,
connectMsg.lPort, connectMsg.fAddress, and connectMsg.fPort.
Start sending and receiving data:
  DoSend();
  DoReceive();
  EXIT();
}
```

Close the TCP connection

This function destroys the shared buffers and closes the connection.

```
void Close()
{
  receiveBuffer.UnMap();
  antEnvDestroySharedBufferMsg
receiveBufferMsg(receiveBuffer);
  receiveBufferMsg.Call(
    IPStackRef,
    sizeof(antEnvDestroySharedBufferMsg)
  );
Do same for the send buffer:
  ...
Then close the connection:
  TCPEndpointCloseMsg closeMsg(connection);
  closeMsg.Call(
    IPStackRef,
    sizeof(TCPEndpointCloseMsg)
  );
}
```

Receive data

These functions receive data until enough data has arrived. Then the TCP connection is closed. Two functions are used, to introduce asynchronisity.

DoReceive() tries to receive between 1 and PACKETSIZE bytes.

```
void DoReceive()
{
  TCPEndpointReceiveMsg receiveMsg(
    connection,
    receiveData,
    1,
    PACKETSIZE
  );
  receiveMsg.Send(
    IPStackRef,
    myOID,
    Entry_ReceiveCont,
    sizeof(TCPEndpointReceiveMsg)
  );
}
```

ReceiveCont() is invoked when a receive request has been completed. If all sent data has been received, the TCP connection is closed—otherwise, a new receive request is posted.

```
void ReceiveCont(ANTENVMSG  _msg)
{
  TCPEndpointReceiveMsg* msg;
  msg = (TCPEndpointReceiveMsg*)antEnvMsg::Receive(_msg);

  if (TCP_SUCCESS == msg->error)
  {
    bytesReceived += msg->sizeMin;
    if (bytesReceived < ECHOSIZE)
    {
      DoReceive();
    } else {
      Close();
    }
  }
  else
  {
    Close();
  }
  EXIT();
}
```

Send data

These functions send data to the echo server. Two functions are used, to introduce asynchronisity.
DoSend() sends PACKETSIZE bytes to the echo server.

```
void DoSend()
{
  TCPEndpointSendMsg sendMsg(
    connection,
    sendData, PACKETSIZE
  );
  sendMsg.Send(
    IPStackRef,
    myOID, Entry_SendCont,
    sizeof(TCPEndpointSendMsg)
  );
}
SendCont() is invoked when the send buffer has been processed
by the IP stack and can be reused by the object. If more data
needs to be sent, it will post another send request.
void SendCont(ANTENVMSG  _msg)
{
  TCPEndpointSendMsg* msg;
  msg = (TCPEndpointSendMsg*)antEnvMsg::Receive(_msg);
  if (TCP_SUCCESS == msg->error)
  {
    bytesSent += msg->size;
    if (bytesSent < ECHOSIZE)
    {
      DoSend();
    }
  }
  else
  {
   Close();
  }
  EXIT();
}
```

3 UDP guide

This chapter introduces the UDP protocol on OPEN-R, and explains how your object can use the UDP services offered by the IPv4 protocol stack.

3.1 Introduction to UDP on OPEN-R

UDP (User Datagram Protocol) is a protocol that runs on top of the IP layer in the IPv4 protocol stack. It forwards packets of data, or datagrams, to the IP layer, which delivers the packets over the network. UDP offers objects a connectionless, unreliable datagram delivery service. Connectionless means that the sending and receiving hosts do not establish a connection.

UDP is used by object-layer protocols such as TFTP, DNS, NFS, and so on.

UDP network operations

On OPEN-R, the IPv4 protocol stack offers the following UDP operations to objects:

- Bind – Set the local connection parameters, which identify the object as a destination for UDP packets. After a bind operation, the object will receive packets only if their destination address is the same as the IP address and port number specified by the bind parameters.
- Connect – An optional operation in which the object sets the foreign connection parameters. Packets will be exchanged only with the host identified by IP address and port number in the connection parameters.
- Send – Send data.
- Receive – Receive data.
- Close – Clear the bind and connect parameters, and stop sending and receiving data.

Your object performs these operations by sending special messages to a UDP endpoint in the IPStack. These messages are inherited from UDPEndpointBaseMessage, which is itself inherited from antEnvMsg. For descriptions of these messages, see “Chapter 8 UDP reference.”

For an overview of how objects create endpoints and request network services, see “1.3 How your object communicates with the protocol stack.”

UDP endpoint life cycle

Figure 5 shows the state transitions of a UDP endpoint during its life cycle. The message types shown in Figure 5 are described fully in “Part2 IPv4 Reference.”

Your object requires one endpoint for each UDP connection, and an endpoint can perform only one similar operation at a time. For example, if you send a UDPEndpointSendMsg to an endpoint that is already sending data, the endpoint will return an UDP_CONNECTION_BUSY error. However, you can send a receive message to this endpoint.

3.2 Creating a UDP endpoint

Before your object can send or receive data by UDP, it must create a new UDP endpoint. Your object requires a new endpoint for each UDP connection that it opens. The process for creating an endpoint is the same for each protocol in the IPStack, and is described in detail in “How your object communicates with the protocol stack.”

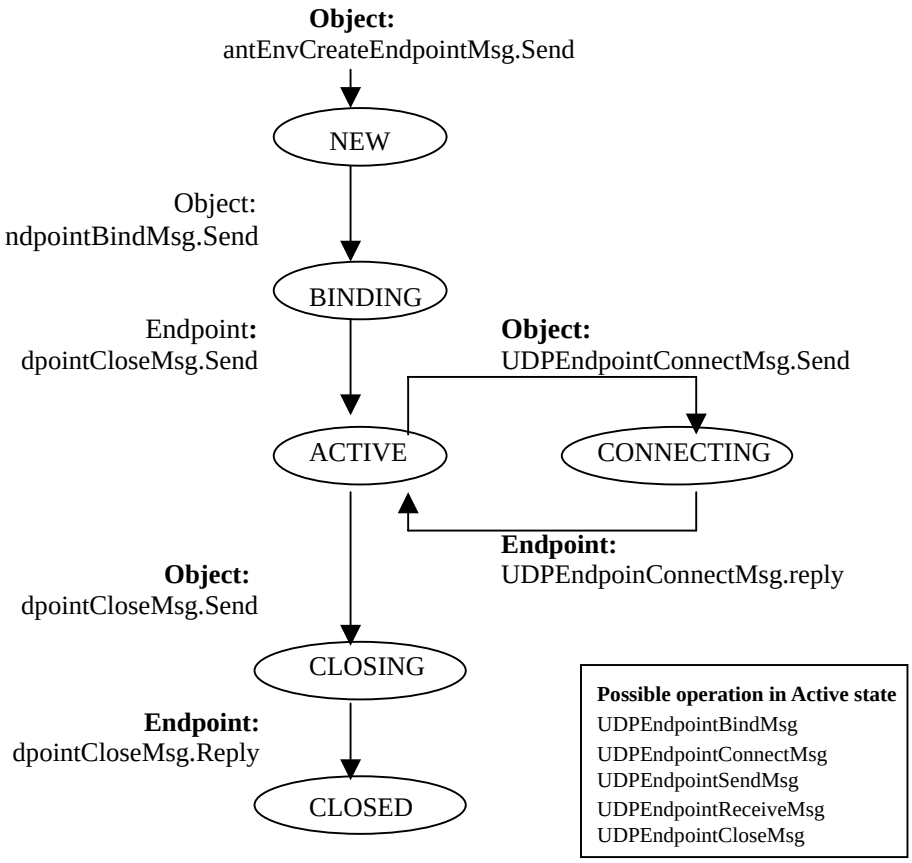

Figure 5 The state transitions of a UDP endpoint.

3.3 Binding an endpoint

An object must bind an endpoint, after the endpoint has been created. Binding is the process of setting the local connection parameters, which identifies the port to be used by the endpoint when sending UDP packets.

To bind an endpoint, your object sends a UDPEndpointBindMsg to the endpoint in the IPStack. Table 5 shows the parameters in this message.

Table 5 Parameters in UDPEndpointBindMsg.

Parameter	Description
address	(in/out) A valid IP address on the local host. If you specify IP_ADDR_ANY, the object will receive packets sent to any IP address on the local host. This is useful for multihomed hosts, which have several interfaces with different addresses. If the host is not multihomed, the local IP address is returned. On a multihomed host, IP_ADDR_ANY will be updated to a specific IP address if the object performs a connect operation (after binding the endpoint.
port	(in/out) The port number of the object. If you specify IP_PORT_ANY, an ephemeral port number is assigned to the object and returned when the endpoint has been bound. This port number will be greater than or equal to 1024.

After an endpoint is bound, the object can send and receive data. Every packet sent by the object must specify a destination IP address and port number, unless the object performs a connect operation first. In a connect operation, the object specifies a destination for all of the packets that it sends. See “3.4 Setting foreign connection parameters (Optional)” for more information.

3.4 Setting foreign connection parameters (Optional)

After binding an endpoint, an object can perform a connect operation. This operation specifies a destination IP address and port number for every packet sent by the object on that endpoint. Once connected, the object no longer needs to specify a destination when it sends a packet.

To perform a connect operation, your object sends a UDPEndpointConnectMsg to its endpoint in the IPStack. Table 5 shows the parameters in this message.

Table 6 Parameters in UDPEndpointConnectMsg.

Parameter	Description
address	(in) Specifies the IP address of the host to which all packets should be sent.
port	(in) Specifies the port number of the object to which all packets should be sent.

3.5 Sending data

To send data by UDP, your object sends a UDPEndpointSendMsg to its endpoint in the IPStack. If the endpoint is bound but not connected, this message must specify a destination IP address and port number. If the endpoint is connected, this information is not required. Table 7 shows the parameters in the UDPEndpointSendMsg message.

Table 7 Parameters in UDPEndpointSendMsg.

Parameter	Description
address	(in) The IP address of the host to which the data should be sent. If your object has performed a connect operation, this parameter is ignored. The IP address specified in the UDPEndpointConnectMsg is used instead.
port	(in) The port number of the object to which the data should be sent. If your object has performed a connect operation, this parameter is ignored. The port number specified in the UDPEndpointConnectMsg is used instead.
buffer	(in) A pointer to the data being sent. This data must be stored in a shared memory buffer, defined by the antSharedBuffer structure.
size	(in) The size of the data being sent, in bytes.

The UDP endpoint replies to this message when the data has been removed from the shared buffer and sent.

3.6 Receiving data

To receive data by UDP, your object sends a UDPEndpointReceiveMsg to its UDP endpoint in the IPStack. Table 8 shows the parameters in this message.

Table 8 Parameters in UDPEndpointReceiveMsg.

Parameter	Description
address	(out) The IP address of the host that sent the data.
port	(out) The port number of the object that sent the data.
buffer	(in) Pointer to the buffer where incoming data should be stored. This data must be stored in a shared memory buffer, defined by the antSharedBuffer structure.
size	(in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received. If the received packet is larger than the specified size, the extra data is deleted.

The UDP endpoint replies to this message when the data has been copied into the shared buffer.

3.7 Closing an endpoint

To close a UDP endpoint, your object sends a UDPEndpointCloseMsg to the endpoint in the IPStack. This message has no parameters. After sending this message, the object can no longer send or receive data. The endpoint will reply when the connection has been fully closed.

3.8 UDP echo server example

This example of a UDP echo server illustrates how to use the UDP messages. A more complete version of this example can be found in the Examples directory of your IPv4 distribution.

The UDP echo server opens a UDP connection and binds it to the echo port (7). All data received on this connection is sent back.
See “Entry point definition” for numbering the entry points.

Variable definitions

```
#include <ant.h>
#include <EndpointTypes.h>
#include <UDPEndpointMsg.h>

enum
{
  Entry_Initialize = 0,
  Entry_ReceiveCont = 1,
  Entry_SendCont = 2,
  NumEntries = 3
};
OPEN-R  OID  and  antStackRef  are  necessary  for  the  message
passing:

OID myOID;
antStackRef IPStackRef;

Maximum UDP packet size:
#define PACKETSIZE 65536

The UDP connection:
antModuleRef connection;

Send/receive buffer:
antSharedBuffer dataBuffer;
byte* data;
```

Entry point definition

The following description is needed for stub.cfg. See “2.5.1 in Programmer’s Guide” for the details of entry points.

```
Extra: Initialize()
Extra: ReceiveCont()
Extra: SendCont()
```

Executing a stubgen2 command generates xxxStub.cc where the following are described.

```
            :
GEN_ENTRY(_Initializestub,      _Initialize);
GEN_ENTRY(_ReceiveContstub,     _ReceiveCont);
GEN_ENTRY(_SendContstub,        _SendCont);

ObjectEntry ObjectEntryTable[]= {
            :
     {Extra_Entry[0],    (Entry)_Initializestub},
     {Extra_Entry[1],    (Entry)_ReceiveCont},
```

```
    {Extra_Entry[2],    (Entry)_SendCont},
    {UNIDEF,            (Entry)_ENTRY_UNDEF}
};
```

The number of Entry points is determined by the order of the elements in ObjectEntryTable[].

Initialize the object

This function is invoked when the client object starts up. It performs the following operations:

- Create a shared buffer for send/receive operations
- Create a UDP endpoint
- Bind the endpoint to the echo port (7)

```
void Initialize(void)
{
Get some OIDs for OPEN-R message passing:
  WhoAmI(&myOID);
  IPStackRef = antStackRef("IPStack");

Allocate a shared buffer for sending/receiving:

  antEnvCreateSharedBufferMsg bufferMsg(PACKETSIZE);
  bufferMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateSharedBufferMsg)
  );
  if (ANT_SUCCESS != bufferMsg.error)
  {
    EXIT();
  }
  dataBuffer = bufferMsg.buffer;
  dataBuffer.Map();

Store the base address of the buffer:
  data = (byte*)dataBuffer.GetAddress();

Create a UDP connection:

antEnvCreateEndpointMsg             createMsg(EndpointType_UDP,
PACKETSIZE);
  createMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateEndpointMsg)
  );
  if (ANT_SUCCESS != createMsg.error)
  {
    EXIT();
  }

  connection = createMsg.moduleRef;

Bind to the echo port:

  UDPEndpointBindMsg bindMsg(connection, IP_ADDR_ANY, 7);
  bindMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateEndpointMsg)
  );

  if (UDP_SUCCESS != bindMsg.error)
  {
```

```
    EXIT();
  }
Start receiving:
  DoReceive();
  EXIT();
}
```

Echo data

These functions receive a UDP datagram together with the IP address and port of the sender, and then send the data back.

DoReceive() posts a receive request for a packet containing up to PACKETSIZE bytes.

```
void DoReceive()
{
  UDPEndpointReceiveMsg receiveMsg(
                                    connection,
                                    dataBuffer, PACKETSIZE
                                   );
  receiveMsg.Send(
    IPStackRef,
    myOID, Entry_ReceiveCont,
    sizeof(UDPEndpointReceiveMsg)
  );
  EXIT();
}

ReceiveCont() is invoked when a UDP datagram has been received.
The function sends the datagram back to the sender of the data.

void ReceiveCont(void* _msg)
{
  UDPEndpointReceiveMsg* receiveMsg;
  receiveMsg =
  (UDPEndpointReceiveMsg*)antEnvMsg::Receive(_msg);

  if (UDP_SUCCESS == receiveMsg->error)
  {
    UDPEndpointSendMsg sendMsg(
                               connection,
                               receiveMsg->address,
                               receiveMsg->port,
                               receiveMsg->buffer,
                               receiveMsg->size
                              );
Back to sender:

    sendMsg.Send(
      IPStackRef,
      myOID, Entry_SendCont,
      sizeof(UDPEndpointSendMsg)
    );

  }
  else
  {
    Close();
  }
  EXIT();
}
```

SendCont() is invoked when the data has been processed by the IP stack and the

buffer can be re-used. It posts a new receive request.

```
void SendCont(ANTENVMSG _msg)
{
  UDPEndpointSendMsg* sendMsg;
  sendMsg =(UDPEndpointSendMsg*)antEnvMsg::Receive(_msg);

  if (UDP_SUCCESS == sendMsg->error)
  {
    DoReceive();
  }

  else
  {
    Close();
  }
  EXIT();
}
```

Close the UDP connection

The Close() function:

- Unmaps and destroys the shared buffer
- Closes the UDP connection

```
void Close()
{
  dataBuffer.UnMap();
  antEnvDestroySharedBufferMsg bufferMsg(dataBuffer);
  bufferMsg.Call(
                 IPStackOID,
                 sizeof(antEnvDestroySharedBufferMsg)
                );

  UDPEndpointCloseMsg closeMsg(connection);
  closeMsg.Call(
                IPStackOID,
                sizeof(UDPEndpointCloseMsg)
               );
}
```

4 DNS guide

This chapter introduces the DNS-client support on OPEN-R, and explains how your object can use the DNS-client services offered by the IPv4 protocol stack.

4.1 Introduction to DNS

DNS (Domain Name System) is a protocol that runs on top of UDP in the IPv4 protocol stack. It offers services for setting, getting, and translating Internet domain names and IP addresses.

For example, if an object wants to send a file using FTP to a host with a domain name of ftpserver.yourdomain.com, the object program must know the IP address of that host. To determine this address, your program sends a request to DNS, which replies that ftpserver.yourdomain.com has an IP address of 192.168.1.200. With this information, your object program can open an FTP connection and begin sending the file to ftpserver.yourdomain.com.

DNS network operations

On OPEN-R, the IPv4 protocol stack offers the following DNS operations to objects:

Set and get a host's DNS servers – Set or get the IP addresses of all DNS servers used by a host. Note that all objects share the DNS server's definition. Therefore, when one object changes the DNS server's definition, all other objects will automatically use that definition.

- Set and get an object's default domain name – Set or get the default domain name of the host.
- Get a complete host entry by domain name or IP address – Get the full list of IP addresses and domain name aliases for a specified host.
- Get a host's IP address – Get any valid IP address for a host, when the host has multiple addresses.
- Get a host's domain name alias – Get any valid alias for a host's domain name, when the domain name has multiple aliases.
- Close – Close an object's DNS endpoint.

Your object performs these operations by sending special messages to a DNS endpoint in the IPStack. These messages are inherited from DNSEndpointBaseMessage, which is itself inherited from antEnvMsg. For descriptions of these messages, see "DNS reference."

For an overview of how objects create endpoints and request network services, see "1.3 How your object communicates with the protocol stack."

DNS endpoint life cycle

Figure 6 shows the state transitions of a DNS endpoint during its life cycle. The message types shown in Figure 6 are described fully in the IPv4 Reference.

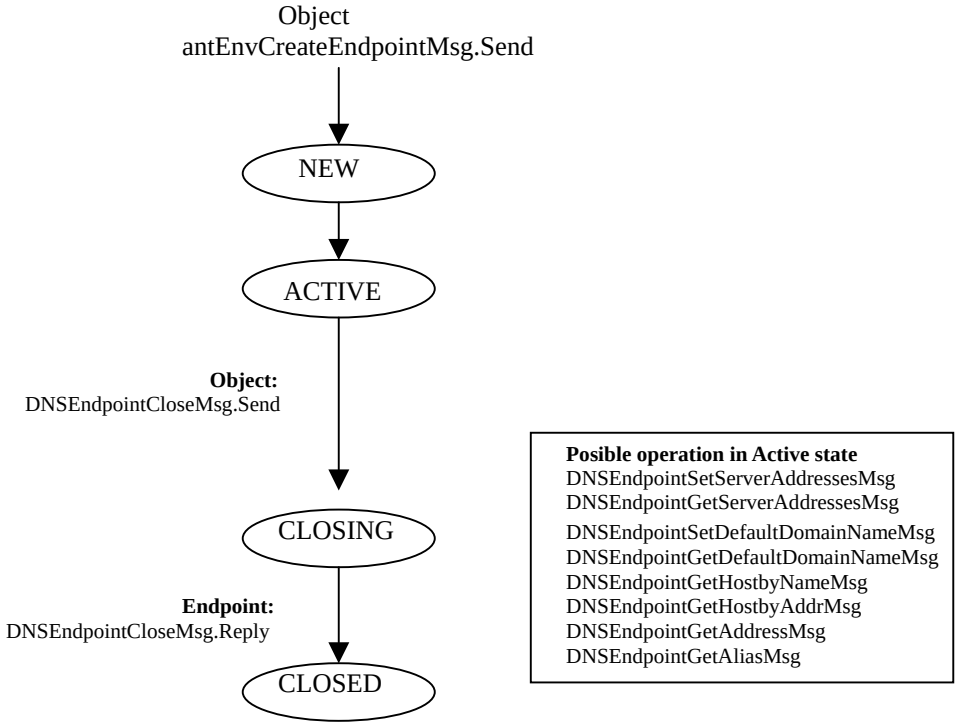

Figure 6 The state transitions of a DNS endpoint

Your object requires one endpoint for each DNS connection, and an endpoint can perform only one operation at a time. For example, if you send a DNSEndpointSetServerAddressesMsg to an endpoint that is already performing that operation, the endpoint will return a DNS_CONNECTION_BUSY error.

4.2 Creating a DNS endpoint

Before your object can send or receive data by DNS, it must create a new DNS endpoint. The process for creating an endpoint is the same for each protocol in the IPStack, and is described in detail in “How your object communicates with the protocol stack.”

If possible, try to open only one DNS endpoint and reuse it throughout the object’s lifespan for all DNS operations and queries. Close the endpoint only when no more DNS queries are expected. Note that it is possible to create multiple DNS endpoints, if your object requires it.

4.3 Setting and getting an object's DNS servers

When an object is first initialized, it should register a list of all DNS servers that it will use for resolving domain names and IP addresses. After this list is registered, queries will be sent to the listed servers in the order they appear in the list.

Setting DNS server IP addresses

To register a list of DNS server IP addresses, your object sends a DNSEndpointSetServerAddressesMsg to its endpoint in the IPStack.
Table 9 shows the parameters in DNSEndpointSetServerAddressesMsg.

Table 9 Parameters in DNSEndpointSetServerAddressesMsg

Parameter	Description
nscount	(in) The number of IP addresses to register.
addrList[MAXNS]	(in) The list of IP addresses to register.

Getting DNS server IP addresses

To get a list of the DNS servers registered for your host, the object sends a DNSEndpointGetServerAddressesMsg to its endpoint in the IPStack.

Table 9 shows the parameters in DNSEndpointGetServerAddressesMsg.

Table 10 Parameters in DNSEndpointGetServerAddressesMsg

Parameter	Description
nscount	(out) The number of registered IP addresses.
addrList[MAXNS]	(out) The list of IP addresses.

4.4 Setting and getting an object's default domain name

When an object is first initialized, it should register its default domain name. After this domain name is registered, the name will automatically be added to all host names that are not fully specified.

Setting the default domain name

To set its default domain name, your object sends a DNSEndpointSetDefaultDomainNameMsg to its endpoint in the IPStack.
DNSEndpointSetDefaultDomainNameMsg has one parameter, which specifies the domain name: name[MAXDNAME]. The domain name must be null-terminated.

Getting the default domain name

To get its default domain name, your object sends a DNSEndpointGetDefaultDomainNameMsg to its endpoint in the IPStack.
DNSEndpointSetDefaultDomainNameMsg has one parameter, which specifies the domain name: name[MAXDNAME]. The domain name is null-terminated.

4.5 Getting a host entry

When you want to determine the IP address, domain name, or domain name aliases of a host on the Internet, your object must get an entry for the host from one of its DNS servers. A host entry typically contains a list of all IP addresses and domain name aliases for the host, as shown in Figure 7.

On OPEN-R, when your object requests a host entry from the DNS protocol, the entry will contain only:

- The **address** of the DNS server that returned the host entry.
- The **first IP address** in the list (the official address), and a count of the total number of IP addresses for the host.
- The **first domain name** in the list (the official domain name), and a count of the total number of domain name aliases for the host.

If you want to get a different IP address or one of the domain name aliases, you must request them specifically. For details, see “Getting a host’s IP address” and “Getting a host’s domain name alias,” later in this chapter.

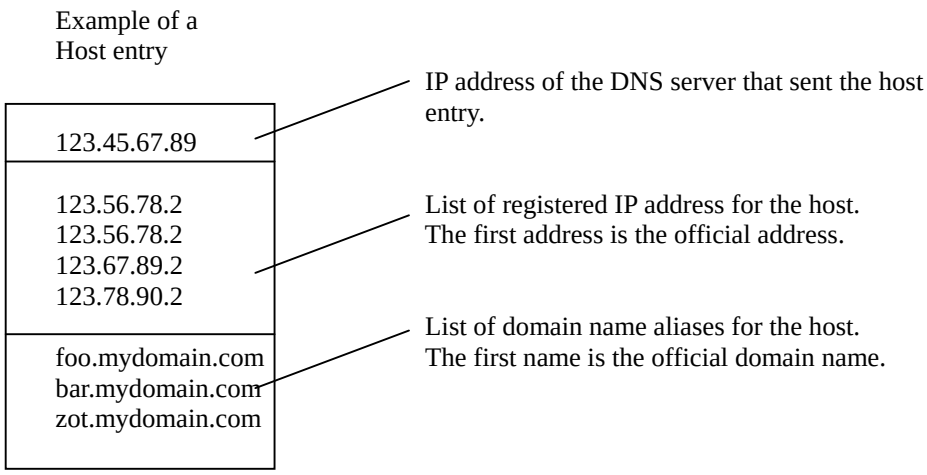

Figure 7 An example of a typical host entry

Getting an entry by domain name

To get a host entry when you know only the domain name or a domain name alias of the host, your object sends a DNSEndpointGetHostByNameMsg to its endpoint in the IPStack.

Table 9 shows the parameters in DNSEndpointGetHostByNameMsg.

Table 11 Parameters in DNSEndpointGetHostByNameMsg

Parameter	Description
name[MAXDNAME]	(in/out) The domain name of the host for which you want to get an entry. The domain name must be null-terminated. If you specify a domain name alias, this parameter will return the official domain name of the host.
server_address	(out) The IP address of the DNS server that sent the host entry.
host_address	(out) The IP address of the host. If the host has more than one IP address, this parameter returns the first IP address.
n_address	(out) The number of IP addresses for the host.
n_alias	(out) The number of domain name aliases for the host.

Getting an entry by IP address

To get a host entry when you know only the IP address of the host, your object sends a DNSEndpointGetHostByAddrMsg to its endpoint in the IPStack.

Table 9 shows the parameters in DNSEndpointGetHostByAddrMsg.

Table 12 Parameters in DNSEndpointGetHostByAddrMsg.

Parameter	Description
host_address	(in/out) The IP address of the host. This parameter returns the first IP address of the host, even if you specify a different IP address in the original message.
server_address	(out) The IP address of the DNS server that sent the host entry.
name[MAXDNAME]	(out) The official domain name of the host. The domain name is null-terminated.
n_address	(out) The number of IP addresses for the host.
n_alias	(out) The number of domain name aliases for the host.

4.6 Getting a host's IP address

If you want to get only the first registered IP address of a host, see "Getting a host entry" earlier in this chapter. If you want to get any other IP addresses registered to the host, you perform an additional operation, described in this section.

To get one of a host's additional IP addresses, your object sends a DNSEndpointGetAddressMsg to its endpoint in the IPStack.

Notes
Before you perform this operation, you must have already received the host entry, as described in "Getting a host entry".

Table 13 shows the parameters in DNSEndpointGetAddressMsg.

Table 13 Parameters in DNSEndpointGetAddressMsg.

Parameter	Description
index	(in) The index of the IP address that you want to get. This index must be less than the value of n_address, returned when the host entry was received earlier. A value of 0 returns the first IP address in the host entry list.
address	(out) The IP address at location index in the host entry.

4.7 Getting a host's domain name alias

If you want to get only the official domain name of a host, see "Getting a host entry" earlier in this chapter. If you want to get any of the host's domain name aliases, you perform an additional operation, described in this section.

To get one of a host's domain name aliases, your object sends a DNSEndpointGetAliasMsg to its endpoint in the IPStack.

Notes
Before you perform this operation, you must have already received the host entry, as described in "4.5 Getting a host entry".

Table 9 shows the parameters in DNSEndpointGetAliasMsg.

Table 14 Parameters in DNSEndpointGetAliasMsg.

Parameter	Description
index	(in) The index of the domain name alias that you want to get. This index must be less than the value of n_alias, returned when the host entry was received earlier. A value of 0 returns the official domain name of the host.
name[MAXDNAME]	(out) The domain name alias at location index in the host entry.

4.8 Closing an endpoint

To close a DNS endpoint, your object sends a DNSEndpointCloseMsg to the endpoint in the IPStack. This message has no parameters. After sending this message, the object can no longer send or receive data.

4.9 DNS client example

This DNS client example illustrates how to use the DNS messages. A complete version of this program can be found in your IPv4 distribution.

The DNS client opens an endpoint to the DNS service in the IP stack, and uses it to look up some names and IP addresses.

Variable definitions

```
#include <ant.h>
#include <EndpointTypes.h>
#include <DNSEndpointMsg.h>

OPEN-R message-passing-related information:
OID myOID;
antStackRef IPStackRef;

The endpoint used to access DNS:
antModuleRef endpoint;

Replace these with local DNS and host values:
#define DNS_SERVER1      “IP of primary DNS server”
#define DNS_SERVER2      “IP of secondary DNS server”
#define DNS_DOMAIN       “local domain name”

#define HOSTNAME1       “www.yahoo.com”
#define HOSTNAME2       “name of local host”

#define HOSTIP1         “IP address of local machine”
```

Entry point definition

The following description is needed for stub.cfg. See “Extra entry, 2.3 Stub in Programmer’s Guide” for the details of the definition of entry points.

```
Extra: Initialize()
```

Initialize the object

This function is invoked when the object starts up.

```
void Initialize ()
{
  WhoAmI(&myOID);
  IPStackRef = antStackRef("IPStack");

  Open();
  SetServers();

  GetHostByName(HOSTNAME1);
  GetHostByName(HOSTNAME2);
  GetHostByAddress(HOSTIP1);

  Close();
  EXIT();
}
```

Open and close endpoint

Open() creates a DNS endpoint, which is needed to communicate with the DNS service in the IP stack.

```
void Open()
{
  antEnvCreateEndpointMsg           createMsg(EndpointType_DNS,
16*1024);
  createMsg.Call(IPStackRef, sizeof(antEnvCreateEndpointMsg));

  if (ANT_SUCCESS != createMsg.error)
  {
    EXIT();
  }

  endpoint = createMsg.moduleRef;
}
```

Close() cleans up the DNS endpoint.

```
void Close()
{
  DNSEndpointCloseMsg closeMsg(endpoint);
  closeMsg.Call(IPStackRef, sizeof(DNSEndpointCloseMsg));

}
```

Set up the DNS functionality

The SetServers() function:

- Sets the DNS primary and secondary servers
- Sets the local domain name

```
void SetServers()
{
  IPAddress addrList[2];

  addrList[0] = DNS_SERVER1;
  addrList[1] = DNS_SERVER2;

  DNSEndpointSetServerAddressesMsg
    setServerMsg(endpoint, 2, addrList);

  setServerMsg.Call(
    IPStackRef,
    sizeof(DNSEndpointSetServerAddressesMsg)
  );


  if (DNS_SUCCESS != setServerMsg.error)
  {
    EXIT();
  }

  DNSEndpointSetDefaultDomainNameMsg
                  setDomainMsg(endpoint, DNS_DOMAIN);
  setDomainMsg.Call(
    IPStackRef,
    sizeof(DNSEndpointSetDefaultDomainNameMsg)
  );

  if (DNS_SUCCESS != setDomainMsg.error)
  {
    EXIT();
  }
}
```

GetHostByName() and GetHostByAddress()

```
void GetHostByName(char* name)
{
  DNSEndpointGetHostByNameMsg getHostMsg(endpoint, name);
  getHostMsg.Call(
    IPStackRef,
    sizeof(DNSEndpointGetHostByNameMsg)
  );

  if (DNS_SUCCESS == getHostMsg.error)
  {
    cout << name << " -> " << getHostMsg.host_address << endl;
  }
  else
  {
    cout << "Could not resolve " << name << endl;
  }
}
void GetHostByAddress(IPAddress address)
```

```
{
  DNSEndpointGetHostByAddrMsg getHostMsg(endpoint, address);
  getHostMsg.Call(
    IPStackRef,
    sizeof(DNSEndpointGetHostByAddrMsg)
  );

  if (DNS_SUCCESS == getHostMsg.error)
  {
    cout << address << " -> " << getHostMsg.name << endl;
  }
  else
  {
    cout << "Could not resolve " << address << endl;
  }
}
```

5 IP Guide

This chapter introduces the IP protocol on OPEN-R, and explains how your object can use the IP services offered by the IPv4 protocol stack.

5.1 Introduction to IP

In the IPv4 protocol stack, the Internet Protocol (IP) layer is responsible for transmitting packets over the network. Typically, OPEN-R objects do not communicate directly with IP. Instead, they open connections with layers on top of IP, such as TCP or UDP. However, some objects may need to use the IP layer directly. For example, if you want to add new protocols without programming in the IPStack, you can write the protocols as OPEN-R objects that communicate directly with the IP layer.

IP network operations

On OPEN-R, the IPv4 protocol stack offers the following IP operations to objects:

- Bind – Bind the IP endpoint to a particular protocol. All packets sent and received by an object over this endpoint will be identified as originating from the specified protocol.
- Send – Send data.
- Receive – Receive data.
- Close – Stop sending and receiving data, and delete the endpoint.

Your object performs these operations by sending special messages to an IP endpoint in the IPStack. These messages are inherited from IPEndpointBaseMessage, which is itself inherited from antEnvMsg. For descriptions of these messages, see “Chapter10 IP reference.”

For an overview of how objects create endpoints and request network services, see “1.3 How your object communicates with the protocol stack.”

IP endpoint life cycle

Figure 8 shows the state transitions of an IP endpoint during its life cycle. The message types shown in Figure 8 are described fully in the IPv4 Reference.

Your object requires one endpoint for each IP connection, and an endpoint can perform only one similar operation at a time. For example, if you send an IPEndpointSendMsg to an endpoint that is already sending data, the endpoint will return a IP_CONNECTION_BUSY error. However, it is possible to send an IPEndpointReceiveMsg to this endpoint.

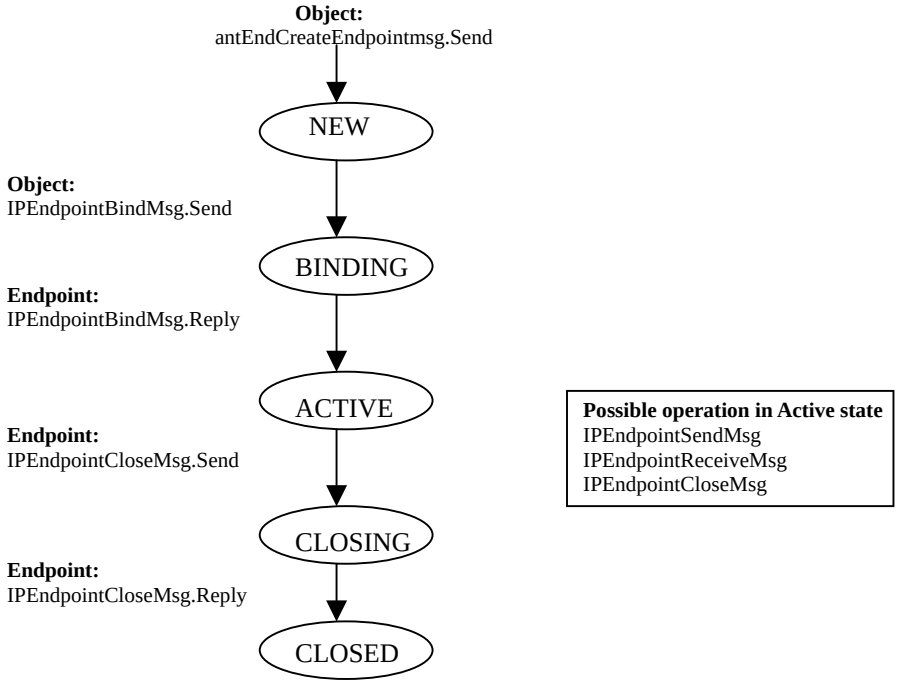

Figure 8 The state transitions of an IP endpoint.

5.2 Creating an IP endpoint

Before your object can send or receive data by IP, it must create a new IP endpoint. The process for creating an endpoint is the same for each protocol in the IPStack, and is described in detail in "1.3 How your object communicates with the protocol stack."

5.3 Binding an endpoint

An object must bind an endpoint to a particular protocol, after the endpoint has been created. All packets sent and received by the object over this endpoint will be identified as originating from the specified protocol.

To bind an endpoint, your object sends an IPEndpointBindMsg to the endpoint in the IPStack. This message has one parameter, called protocol, which specifies the protocol to bind to the endpoint. Each protocol is identified by an integer with a value less than 256.

Notes
It is not possible to bind an endpoint to the TCP or UDP protocols unless the endpoint has already been bound to the protocol stack. If attempted, the error IP_INVALID_PROTOCOL will be returned.

However, it is possible to bind an endpoint to ICMP. ICMP will process all packets that it recognizes, as usual, but will forward all unidentified packets to the endpoint that you have bound.

5.4 Sending data

To send data by IP, your object sends an IPEndpointSendMsg to its endpoint in the IPStack. Table 15 shows the parameters in the UDPEndpointSendMsg message.

Table 15 Parameters in IPEndpointSendMsg

Parameter	Description
type	(in) The type of packet being sent. Normal IP packets are of type IP_DATA. Other types exist for ICMP packets.
buffer	(in) Pointer to the packet being sent. This packet must be stored in a shared memory buffer, defined by the antSharedBuffer structure.
size	(in) The size of the packet being sent, in bytes.

The IP endpoint replies to this message when the data has been removed from the shared buffer and sent.

When sending raw IP packets, you have to fill in the complete IP packet. The IP header is shown in Figure 9. A C++ class, IPHeader, is defined in the IPProtocol.h file. Use the IPHeader to fill in the raw IP packet. The class will take care of all endian-related issues.

It is mandatory to fill in the following fields

- 8-bit protocol field [IPHeader::ip_pSet(byte _val)]
- 32-bit source IP address [IPHeader::ip_srcSet(uint32 _val)]
- 32-bit destination IP address [IPHeader::ip_dstSet(uint32 _val)]

The following fields are optional. When set to zero, they are considered unspecified and will be overwritten by the stack:

- 16-bit total length (in bytes) [IPHeader::ip_lenSet(uint16 _val)]
- 4-bit header length [IPHeader::ip_hlSet(byte_val)]
- 8-bit type of service [IPHeader::ip_tosSet(byte _val)]
- 8-bit time to live [IPHeader::ip_ttlSet(byte _val)]

The time to live field is overwritten by the stack only when the IP packet is not a broadcast packet.

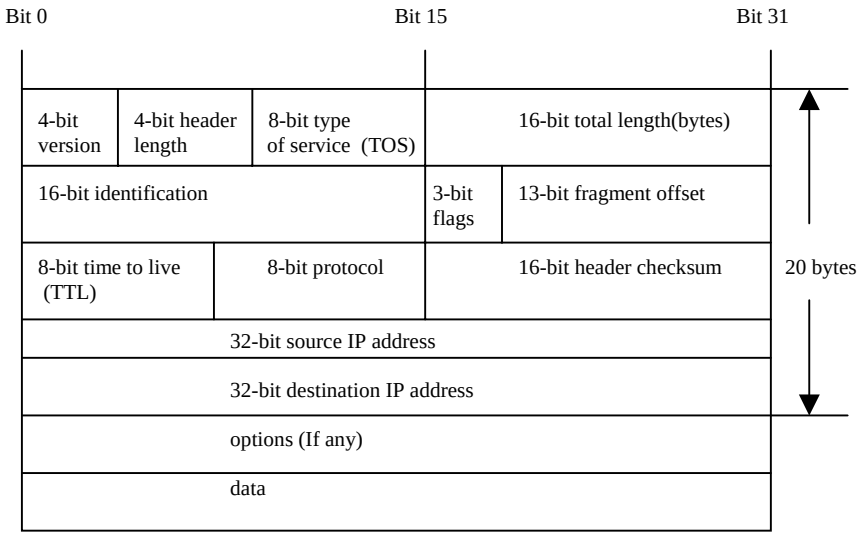

Figure 9 Contents of the IP header

The following fields should not be set, because they are always overwritten by the IP stack:

- 4-bit version
- 13-bit fragment offset
- 16-bit identification
- 16-bit header checksum

Also see “5.7 IP-ping example” at the end of this chapter, for an example of how to use the IPHeader class.

5.5 Receiving data

To receive data by IP, your object sends an IPEndpointReceiveMsg to its endpoint in the IPStack. Table 16 shows the parameters in this message.

Table 16 Parameters in IPEndpointReceiveMsg

Parameter	Description
type	(out) The type of packet received. Normal IP packets are of type IP_DATA. Other types exist for ICMP packets.
buffer	(in) Pointer to an area where the packet being received should be stored. This area must be in a shared memory buffer, defined by the antSharedBuffer structure.
size	(in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received. If the packet being received is too big for the shared buffer, only part of the packet will be stored in the buffer. The error IP_PACKETSIZE error will be returned.

The IP endpoint replies to this message when the data has been copied into the shared buffer.

5.6 Closing an endpoint

To close an IP endpoint, your object sends an IPEndpointCloseMsg to the endpoint in the IPStack. This message has no parameters. After sending this message, the object can no longer send or receive data. The endpoint will reply when the connection has been fully closed.

5.7 IP ping example

This IP ping example illustrates how to use the IP messages.
The IP ping program sends an ICMP packet to a host on the network and waits for a reply.

Variable definitions

```
#include <ant.h>
#include <EndpointTypes.h>
#include <IPEndpointMsg.h>

#include <IPProtocol.h>
#include <ICMPProtocol.h>
#include <endian.h>
Message-passing-related information:

OID myOID;
antStackRef IPStackRef;
IP endpoint:

antModuleRef connection;

Buffer used to send packet:
antSharedBuffer dataBuffer;
byte* data;

Destination for ping packet:
#define HOSTIP  “192.168.1.2”
```

Entry point definition

The following description is needed for stub.cfg. See “Extra entry, 2.3 Stub in Programmer’s Guide” for the details of the definition of entry points.

```
Extra : Initialize()
```

Initialize the object

This function is invoked when the object starts up.

```
void Initialize()
{
  WhoAmI(&myOID);
  IPStackRef = antStackRef(“IPStack”);

  Open();
  Ping(HOSTIP);
  Close();

  EXIT();
}
```

Open endpoint

The Open() function:

- Creates a shared buffer for sending and receiving ICMP packets
- Creates an IP endpoint
- Binds this endpoint to the ICMP protocol

```
void Open()
{
Allocate a shared buffer:
  antEnvCreateSharedBufferMsg bufferMsg(1024);
  bufferMsg.Call(
    IPStackRef,
    sizeof(antEnvCreateSharedBufferMsg)
  );

  if (ANT_SUCCESS != bufferMsg.error)
  {
    EXIT();
  }

  dataBuffer = bufferMsg.buffer;
  dataBuffer.Map();

Store the base address of the buffer:
  data = (byte*)dataBuffer.GetAddress();

Create an IP endpoint:
  antEnvCreateEndpointMsg createMsg(EndpointType_IP, 4096);
  createMsg.Call(
    IPStackOID,
    sizeof(antEnvCreateEndpointMsg)
  );

  if (ANT_SUCCESS != createMsg.error)
  {
    EXIT();
  }

  connection = createMsg.moduleRef;

Bind to the ICMP protocol:
  IPEndpointBindMsg bindMsg(connection, IPPROTO_ICMP);
  bindMsg.Call(
    IPStackRef,
    sizeof(IPEndpointBindMsg)
  );

  if (IP_SUCCESS != bindMsg.error)
  {
    EXIT();
  }
}
```

Close

The Close() function:

- Unmaps and destroys the shared buffer
- Destroys the IP endpoint

```
void Close()
{

Destroy the shared buffer:
  dataBuffer.UnMap();
  antEnvDestroySharedBufferMsg bufferMsg(dataBuffer);
  bufferMsg.Call(
    IPStackRef,
    sizeof(antEnvDestroySharedBufferMsg)
  );

Close the endpoint:
  IPEndpointCloseMsg closeMsg(connection);
  closeMsg.Call(
    IPStackRef,
    sizeof(IPEndpointCloseMsg)
  );
}
```

Ping

Ping() sends and receives the actual ICMP_ECHO packet.
It first creates a raw IP/ICMP packet in the shared buffer, which is then put on the network by the IP stack. After this, Ping() waits for an ICMP_ECHOREPLY packet to be received.

```
void Ping(char* host)
{
  IPAddress dest(host);
Pointers to packet headers:
  IPHeader* ipheader = (IPHeader*)data;
  ICMPEcho* icmpecho = (ICMPEcho*)(data+sizeof(IPHeader));

  cout << “Pinging “ << dest << endl;

Fill IP header:
  ipheader->ip_srcSet(IP_ADDR_ANY);
  ipheader->ip_dstSet(dest);
  ipheader->ip_tosSet(0);
  ipheader->ip_ttlSet(IPDEFTTL);
  ipheader->ip_pSet(IPPROTO_ICMP);
  ipheader->ip_offSet(0);

Fill ICMP header:
  icmpecho->icmp_typeSet(ICMP_TYPE_ECHO);
  icmpecho->icmp_cksumSet(0);
  icmpecho->icmp_idSet(0xaf);
  icmpecho->icmp_seqSet(1);

Calculate ICMP checksum:
  uint32 cksum = 0;
  int size16 = sizeof(ICMPEcho) / sizeof(uint16);
  for (int index = 0; index < size16; index++)
  {
    cksum += ((uint16*)icmpecho)[index];
  }
```

```
  while (cksum > 0xffff)
  {
    cksum = (cksum & 0xffff) + (cksum >> 16);
  }

  icmpecho->icmp_cksumSet(htons(~(cksum & 0xffff)));

Send the packet to IP:
  IPEndpointSendMsg sendMsg(
    connection,
    IP_DATA,
    data,
    sizeof(IPHeader) + sizeof(ICMPEcho)
  );
  sendMsg.Call(
    IPStackRef,
    sizeof(IPEndpointSendMsg)
  );

Wait for an answer:
  IPEndpointReceiveMsg receiveMsg(
    connection,
    data, 1024
  );
  receiveMsg.Call(
    IPStackRef,
    sizeof(IPEndpointReceiveMsg)
  );

Check packet:
  if (ICMP_TYPE_ECHOREPLY == icmpecho->icmp_typeGet())
  {
    cout << IPAddress(ipheader->ip_srcGet()) << " is alive" <<
endl;
  }
}
```

Part2 IPv4 Reference
6 ANT environment reference

This chapter describes the structures and methods that you require when sending requests for endpoints and shared buffers to the ANT environment.

antEnvCreateEndpointMsg

```
struct antEnvCreateEndpointMsg: public antEnvMsg
{
public:
      antError error;
      int32 protocol;
      int32 poolSize;
      antModuleRef moduleRef;

public:
      // constructors
      antEnvCreateEndpointMsg(): antEnvMsg() {};
      antEnvCreateEndpointMsg(int32 _protocol,
            int32 _poolSize = 0);

// reply
      antError Reply(antError _error);
};
```

Description
Defines the message that requests a new endpoint.

When an object requires a new endpoint, it creates an antEnvCreateEndpointMsg that specifies what protocol the endpoint is needed for and what size of SDU pool should be created for the endpoint.

Parameters

error (out) Returns an error that describes the result of the request.

protocol (in) The type of endpoint to create, which corresponds to the available protocols

EndpointType_TCP
EndpointType_UDP
EndpointType_TFTP
EndpointType_DNS
EndpointType_POP3
EndpointType_SMTP
EndpointType_IP
EndpointType_MIBII
EndpointType_MIB_ETHERNET

poolSize (in) The size of the SDU pool to create for the new endpoint.
An SDU pool is an internal ANT construct that stores data in the protocol stack.
As a guideline, always create an SDU pool that is slightly larger than the largest packet that you expect to send. For example, 8-KB packets would require an SDU pool of approximately 10 KB.

moduleRef (out) A reference to the new endpoint.

antEnvCreateEndpointMsg::antEnvCreateEndpointMsg()

Constructor

```
antEnvCreateEndpointMsg(
      int32 _protocol,
      int32 _poolSize = 0
);
```

Description
Creates an instance of antEnvCreateEndpointMsg.

When the message has been created, it must be sent to the ANT environment. When the object receives a reply, the moduleRef parameter of antEnvCreateEndpointMsg contains a reference to the new endpoint.

Parameters

_protocol (in) The type of endpoint to create, which corresponds to the available protocols

EndpointType_TCP
EndpointType_UDP
EndpointType_TFTP
EndpointType_DNS
EndpointType_POP3
EndpointType_SMTP
EndpointType_IP
EndpointType_MIBII
EndpointType_MIB_ETHERNET

_poolSize (in) The size of the SDU pool to create for the new endpoint.

antEnvCreateSharedBufferMsg

Structure

```
struct antEnvCreateSharedBufferMsg: public antEnvMsg
{
public:
      antError error;
      uint32 size;
      antSharedBuffer buffer;
};
```

Description
Defines the message that requests a shared memory buffer.

When an object requires a new shared memory buffer, it creates an antEnvCreateSharedBufferMsg that specifies the size of the buffer.

Parameters

error (out) Returns an error that describes the result of the request.

size (in) The size of the shared buffer, in bytes.

buffer (out) A reference to the new buffer, after it has been created.

antEnvCreateSharedBufferMsg::antEnvCreateSharedBufferMsg()

Constructor

---

antEnvCreateSharedBufferMsg(uint32 _size);

Description
Creates an instance of antEnvCreateSharedBufferMsg.

When the message has been created, it must be sent to the ANT environment. When the object receives a reply, the buffer parameter of the antEnvCreateSharedBufferMsg structure returns a reference to the new buffer.

Parameters

size (in) The size of the shared buffer, in bytes.

antSharedBuffer

Member

```
class antSharedBuffer
{
public:
      antError Map();
      antError UnMap();
      void* GetAddress();
      uint32 GetSize();
};
```

Description
Defines the shared memory buffers used to exchange data between OPEN-R applications and the ANT environment.

A shared memory buffer maps a common memory area into the address spaces of an object and the ANT environment. When the object exchanges data with the ANT environment, the data is identified by a pointer into this shared buffer. antSharedBuffer can convert this pointer between the application and the ANT environment address spaces.

To create a shared buffer, an object sends an antEnvCreateSharedBufferMsg to the ANT environment, specifying the size of the buffer.

antSharedBuffer::Map()

Member

---

antError Map();

Description
Maps a shared memory buffer to the address space of an object.
This operation is required before the object can exchange data with the ANT environment.

Return codes

ANT_SUCCESS	**Success**
ANT_FAIL	**Failure**

antSharedBuffer::UnMap()

Member

---

antError UnMap();

Description
Removes a shared memory buffer from an object's address space.
This operation is required before the shared buffer can be destroyed.

Returned value

ANT_SUCCESS	**Success**
ANT_FAIL	**Failure**

antSharedBuffer::GetAddress()

Member

---

void* GetAddress();

Description
Gets the base address of a shared memory buffer.
Before performing this operation, the object must have mapped the buffer to its address space (see antSharedBuffer::Map()).

Returned value

0	**Success**

antSharedBuffer::GetSize()

Member

---

uint32 GetSize();

Description
Gets the size of a shared memory buffer, in bytes.

Before performing this operation, the object must have mapped the buffer to its address space (see antSharedBuffer::Map()).

7 TCP reference

This chapter describes the TCP messages that an object sends to the IPstack to request TCP services. All messages are inherited from TCPEndpointBaseMsg.

TCP errors

Table 17 describes the errors that can be returned by any of the messages in this chapter. The possible error values are defined in the TCPEndpointError enumerated type.

When your object requests a network operation and receives a reply from the TCP endpoint, it should examine the request's error field (for example, TCPEndpointListenMsg.error).

Table 17 TCPEndpointError

Error Value	Description
TCP_BUFFER_INVALID	The address and size parameters provided in a TCPEndpointSendMsg or TCPEndpointReceiveMsg do not fall within a shared buffer.
TCP_CONNECTION_BUSY	The connection is busy and the requested operation cannot be completed. The TCP endpoint to which you sent the message may already be processing another request.
TCP_CONNECTION_CLOSED	The connection has been closed.
TCP_CONNECTION_RESET	The connection has been aborted.
TCP_CONNECTION_TIMEOUT	The connection timed out, and has been closed.
TCP_FAIL	The operation failed (no more information is available).
TCP_HOST_UNREACHABLE	The IP stack was unable to find a route to the destination address.
TCP_MESSAGE_TOO_LONG	An intermediate router was unable to process a TCP/IP packet because it was too big.
TCP_NETWORK_UNREACHABLE	The IP stack was unable to find a route to the network containing the destination address.
TCP_OPERATION_INVALID	The requested operation is not allowed in the current state of the endpoint.
TCP_OPERATION_UNKNOWN	The requested operation was not recognized by the TCP endpoint.
TCP_PORT_UNREACHABLE	There is nobody listening on the destination port specified for the connection.

TCP_PROTOCOL_UNREACHABLE	The destination host is not running a TCP implementation.
TCP_SUCCESS	Operation succeeded.
TCP_TIME_EXCEEDED	An IP reassembly queue timed out.
TCP_TTL_EXCEEDED	The destination host is more than TTL hops from the source.

TCPEndpointBaseMsg

Definition

```
struct TCPEndpointBaseMsg: public antEnvMsg
{
public:
  TCPEndpointError error;

public:
  TCPEndpointBaseMsg() : error(TCP_FAIL), antEnvMsg() {};
  TCPEndpointBaseMsg(
    antModuleRef& _module,
    TCPEndpointOperation _operation
  );
};
```

Description
Specify which endpoint in the IPv4 protocol stack should receive a request for a TCP service, and which OPEN-R object should receive the reply for the request.

Requests for specific network services are sent in messages inherited from TCPEndpointBaseMsg, and are described later in this chapter.

Parameters

module	(in) The target endpoint.
operation	(in) The operation requested by the sending object.
error	(out) See “TCP error” for a description of the possible TCP error codes.

Returned value
See “TCP error” for a description of the possible TCP error codes.

See also
`TCPEndpointConnectMsg, TCPEndpointListenMsg, TCPEndpointSendMsg, TCPEndpointReceiveMsg, TCPEndpointCloseMsg`

TCPEndpointConnectMsg

Definition

```
struct TCPEndpointConnectMsg: public TCPEndpointBaseMsg
{
public:
  IPAddress lAddress;
  Port lPort;
  IPAddress fAddress;
  Port fPort;

public:
  TCPEndpointConnectMsg() : TCPEndpointBaseMsg() {};
  TCPEndpointConnectMsg(
    antModuleRef& module,
    IPAddress lAddress, Port lPort,
    IPAddress fAddress, Port fPort
  );
};
```

Description
Open a TCP connection to another host.

TCPEndpointConnectMsg is normally sent by client objects. The TCP endpoint replies to this message when the connection has been fully established. The reply holds the fully specified local and foreign addresses and port numbers.

TCPEndpointConnectMsg is inherited from TCPEndpointBaseMsg.

Parameters

module	Destination module reference
lAddress	(out) Returns the local IP address, when the connection has been established.
lPort	(out) Returns an ephemeral port number assigned to the client object, when the connection has been established.
fAddress	(in) The IP address of the computer that you need to connect to.
fPort	(in) The port number of the object that you need to connect to.

Returned value
See “TCP error” for a description of the possible TCP error codes.

See also
`TCPEndpointBaseMsg`

TCPEndpointListenMsg

Definition

```
struct TCPEndpointListenMsg: public TCPEndpointBaseMsg
{
public:
  IPAddress lAddress;
  Port lPort;
  IPAddress fAddress;
  Port fPort;

public:
  TCPEndpointListenMsg() : TCPEndpointBaseMsg() {};
  TCPEndpointListenMsg(
    antModuleRef& _module,
    IPAddress lAddress,
    Port lPort,
    IPAddress fAddress = 0,
    Port fPort = 0
  );
};
```

Description
Start listening for connection requests.

TCPEndpointListenMsg is normally sent by server objects. The TCP endpoint replies to this message when the connection has been fully established. The reply holds the fully specified local and foreign addresses and port numbers.

It is possible for a server object to perform more than one listen operation with all accepting connection requests made to the same port number. The object requires a separate endpoint for each listen operation.

TCPEndpointListenMsg is inherited from TCPEndpointBaseMsg.

Parameters

module Destination module reference

lAddress (out) Returns the local IP address, when a connection has been established.

lPort (in) The port number that you will accept connection requests. If you will accept requests for any port, specify a value of IP_PORT_ANY.

fAddress (out) Returns the IP address of the computer that requested the connection.

fPort (out) Returns the port number of the object that requested the connection

.

Returned value
See “TCP error” for a description of the possible TCP error codes.

See also
`TCPEndpointBaseMsg`

TCPEndpointSendMsg

Constructor

```
struct TCPEndpointSendMsg: public TCPEndpointBaseMsg
{
public:
  byte* buffer;
  int size;
public:
  TCPEndpointSendMsg() : TCPEndpointBaseMsg() {};
  TCPEndpointSendMsg(
    antModuleRef& module,
    byte* buffer,
    int size
  );
};
```

Description
Send data over an open TCP connection.

Any data that your object sends to the protocol stack must be stored in a shared memory buffer, defined by the antSharedBuffer structure.

The TCP endpoint replies to this message when the data is copied from the shared buffer to the IP stack internal memory buffers.

TCPEndpointSendMsg is inherited from TCPEndpointBaseMsg.

Parameters

module Destination module reference

buffer (in) The shared buffer where the data being sent is stored.

size (in) The size of the data being sent, in bytes.

Returned value
See “TCP error” for a description of the possible TCP error codes.

See also
`TCPEndpointBaseMsg, TCPEndpointConnectMsg, antSharedBuffer`

TCPEndpointReceiveMsg

Constructor

```
struct TCPEndpointReceiveMsg: public TCPEndpointBaseMsg
{
public:
  byte* buffer;
  int sizeMin;
  int sizeMax;

public:
  TCPEndpointReceiveMsg() : TCPEndpointBaseMsg() {};
  TCPEndpointReceiveMsg(
    antModuleRef& _module,
    byte* buffer,
    int sizeMin,
    int sizeMax
  );
};
```

Description
Receive data from an open TCP connection.

Any data that your object receives from the protocol stack is stored in a shared memory buffer, defined by the antSharedBuffer structure.

The TCP endpoint replies to this message when the data has been copied into the shared buffer. When all data in the transmission has been received and the TCP connection is closed, the last receive request may hold a smaller number of bytes than what is specified in sizeMin.

TCPEndpointReceiveMsg is inherited from TCPEndpointBaseMsg.

Parameters

module Destination module reference

buffer (in) The shared buffer where the data being received should be stored.

sizeMin (in/out) Specifies the minimum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received.

sizeMax (in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received.

Returned value
See “TCP error“for a description of the possible TCP error codes.

See also
`TCPEndpointBaseMsg, TCPEndpointConnectMsg, antSharedBuffer`

TCPEndpointCloseMsg

Constructor

```
struct TCPEndpointCloseMsg: public TCPEndpointBaseMsg
{
public:
  boolean abort;

public:
  TCPEndpointCloseMsg() : TCPEndpointBaseMsg() {};
  TCPEndpointCloseMsg(
    antModuleRef& module,
    boolean abort = FALSE
  );
};
```

Description
Close a TCP connection.
A TCP connection can be closed three different ways:

- **Active close** – The close request is sent directly by your object. The object on the other side of the connection will receive the rest of the transmission, then will receive an error to indicate that you have closed the connection. From the perspective of the other object, a passive close has occurred.

- **Passive close** – The close request is sent by the object on the other side of the connection. After your object has received the entire data transmission, a TCP_CONNECTION_CLOSED error will occur. Your object must then complete the passive close by sending a TCPEndpointCloseMsg to its endpoint in the ANT environment.

- **Abort** – An error occurs, which closes the connection unexpectedly. An abort purges all data from the shared buffers and immediately closes the connection.

The TCP endpoint replies to this message when the connection has been fully closed.

TCPEndpointCloseMsg is inherited from TCPEndpointBaseMsg.

Parameters

module	Destination module reference
abort	(in) If TRUE, the TCP connection is aborted, instead of being shut down in an orderly fashion.

Returned value
See “TCP error” for a description of the possible TCP error codes.

See also
`TCPEndpointBaseMsg`

8 UDP reference

This chapter describes the UDP messages that an object sends to the IPstack, to request UDP services. All messages are inherited from UDPEndpointBaseMsg.

UDP errors

Table 18 describes the errors that can be returned by any of the messages in this chapter. The possible error values are defined in the UDPEndpointError enumerated type.

When your object requests a network operation and receives a reply from its UDP endpoint, it should examine the request's error field (for example, UDPEndpointConnectMsg.error).

Table 18 UDPEndpointError

Error Value	Description
UDP_ADDRESSERROR	The IP address/port number combination that you specified is not valid.
UDP_ADDRESSINUSE	The IP address/port number combination that you specified is already used by another connection.
UDP_BUFFER_INVALID	The address and size parameters provided in a TCPEndpointSendMsg or TCPEndpointReceiveMsg do not fall within a shared buffer.
UDP_CONNECTION_BUSY	The connection is busy and the requested operation cannot be completed. The UDP endpoint to which you sent the message may already be processing another request.
UDP_CONNECTION_CLOSED	The endpoint has been closed.
UDP_FAIL	The operation failed (no more information is available).
UDP_HOST_UNREACHABLE	The IP stack was unable to find a route to the destination address.
UDP_MESSAGE_TOO_LONG	An intermediate router was unable to process a TCP/IP packet because it was too big.
UDP_NETWORK_UNREACHABLE	The IP stack was unable to find a route to the network containing the destination address.
UDP_OPERATION_INVALID	The requested operation is not allowed in the current state of the endpoint.
UDP_OPERATION_UNKNOWN	The requested operation was not recognized by the UDP endpoint.
UDP_PORT_UNREACHABLE	There is nobody listening on the destination port specified for the connection.

UDP_PROTOCOL_UNREACHABLE	The destination host is not running a UDP implementation.
UDP_SUCCESS	Operation succeeded.
UDP_TIME_EXCEEDED	An IP reassembly queue timed out.
UDP_TTL_EXCEEDED	The destination host is more than TTL hops from the source.

UDPEndpointBaseMsg

Definition

```
struct UDPEndpointBaseMsg: public antEnvMsg
{
  public:
    UDPEndpointError error;

  public:
    UDPEndpointBaseMsg() : error(UDP_FAIL), antEnvMsg() {};
    UDPEndpointBaseMsg(
      antModuleRef& module,
      UDPEndpointOperation operation
    );
};
```

Description
Specify which endpoint in the IPv4 protocol stack should receive a request for a UDP service, and which OPEN-R object should receive the reply for the request.

Requests for specific network services are sent in messages inherited from UDPEndpointBaseMsg, and are described later in this chapter.

Parameters

error (out)UDPEndpointError returned by the endpoint. See Table 18 for the list of errors.

module (in)The target endpoint.

operation (in)The operation requested by the sending object.

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBindMsg, UDPEndpointConnectMsg, UDPEndpointSendMsg, UDPEndpointReceiveMsg, UDPEndpointCloseMsg`

UDPEndpointBindMsg

Definition

```
struct UDPEndpointBindMsg: public UDPEndpointBaseMsg
{
public:
  IPAddress address;
  Port port;

public:
  UDPEndpointBindMsg() : UDPEndpointBaseMsg() {};
  UDPEndpointBindMsg(
    antModuleRef& module,
    IPAddress address,
    Port port
  );
};
```

Description
Set the local connection parameters, which identify the object as a destination for UDP packets.

After a bind operation, the object receives packets if the destination address and port are the same as the IP address and port number specified by the bind parameters. When sending data, every packet must specify a destination IP address and port number, unless the object performs a connect operation first. In a connect operation, the object specifies a destination for all packets that it sends. See UDPEndpointConnectMsg for more information.

UDPEndpointBindMsg is inherited from UDPEndpointBaseMsg.

Parameters

module Destination module reference

address (in/out) A valid IP address on the local host.
If you specify IP_ADDR_ANY, the object will receive packets sent to any IP address on the local host. This is useful for multihomed hosts, which could have several interfaces with different addresses. If the host is not multihomed, the local IP address is returned.
On a multihomed host, IP_ADDR_ANY is updated to a specific IP address if the object performs a connect operation after binding.

port (in/out) The port number of the object.
If you specify IP_PORT_ANY, an ephemeral port number is assigned to the object and returned when the endpoint has been bound. This port number will be greater than or equal to 1024.

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBaseMsg, UDPEndpointConnectMsg`

UDPEndpointConnectMsg

Definition

```
struct UDPEndpointConnectMsg: public UDPEndpointBaseMsg
{
public:
  IPAddress address;
  Port port;

public:
  UDPEndpointConnectMsg() : UDPEndpointBaseMsg() {};
  UDPEndpointConnectMsg(
    antModuleRef& module,
    IPAddress address,
    Port port
  );
};
```

Description
Specify a destination IP address and port number for every packet sent by the object.

This operation must be performed after sending a UDPEndpointBindMsg. Once connected, an object no longer needs to specify a destination when it sends a packet.

UDPEndpointConnectMsg is inherited from UDPEndpointBaseMsg.

Parameters

module	Destination module reference
address	(in) Specifies the IP address of the computer to which all packets should be sent.
port	(in) Specifies the port number of the object to which all packets should be sent.

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBaseMsg, UDPEndpointBindMsg`

UDPEndpointSendMsg

Definition

```
struct UDPEndpointSendMsg: public UDPEndpointBaseMsg
{
public:
  IPAddress address;
  Port port;
  byte* buffer;
  int size;

public:
  UDPEndpointSendMsg() : UDPEndpointBaseMsg() {};
  UDPEndpointSendMsg(
    antModuleRef& module,
    byte* buffer,
    int size
  );

  UDPEndpointSendMsg(
    antModuleRef& module,
    IPAddress address,
    Port port,
    byte* buffer,
    int size
 );
};
```

Description
Send data through a UDP endpoint.

Any data that your object sends to the protocol stack must be stored in a shared memory buffer, defined by the antSharedBuffer structure.

The UDP endpoint replies to this message when the data has been copied from the shared buffer into the IPStack internal memory buffers.
UDPEndpointSendMsg is inherited from UDPEndpointBaseMsg.

Notes
If the endpoint is bound but not connected, this message must specify a destination IP address and port number. If the endpoint has been connected by a UDPEndpointConnectMsg, this information is not required.

Parameters

module Destination module reference

address (in) The IP address of the computer to which the data should be sent. If your object has performed a connect operation, this parameter is ignored. The IP address specified in the UDPEndpointConnectMsg is used instead.

port (in) The port number of the object to which the data should be sent. If your object has performed a connect operation, this parameter is ignored. The port number specified in the UDPEndpointConnectMsg is used instead.

buffer (in) Location (in a shared buffer) where the data being sent is stored.

size (in) The size of the data being sent, in bytes.

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBaseMsg, UDPEndpointConnectMsg, antSharedBuffer`

UDPEndpointReceiveMsg

Definition

```
struct UDPEndpointReceiveMsg: public UDPEndpointBaseMsg
{
public:
  IPAddress address;
  Port port;
  byte* buffer;
  int size;
public:
  UDPEndpointReceiveMsg() : UDPEndpointBaseMsg() {};
  UDPEndpointReceiveMsg(
    antModuleRef& module,
    byte* buffer,
    int size
  );
};
```

Description
Receive data from a UDP endpoint.

Any data that your object receives from the protocol stack is stored in a shared memory buffer, defined by the antSharedBuffer structure.

The UDP endpoint replies to this message when the data has been copied into the shared buffer. When the receive operation has completed, the size parameter returns the actual number of bytes that were received. If the received packet is larger than the specified size, the extra data is deleted.

UDPEndpointReceiveMsg is inherited from UDPEndpointBaseMsg.

Parameters

module Destination module reference

buffer (in) The location (in a shared buffer) where the data being received should be stored.

size (in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed, this parameter returns the actual number of bytes received. If the received packet is larger than size, the extra data is deleted.

address (out) IP address from where the received data originated

port (out) port from which the received data originated

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBaseMsg, UDPEndpointConnectMsg, antSharedBuffer`

UDPEndpointCloseMsg

Definition

```
struct UDPEndpointCloseMsg: public UDPEndpointBaseMsg
{
public:
  UDPEndpointCloseMsg() : UDPEndpointBaseMsg() {};
  UDPEndpointCloseMsg(antModuleRef& module);
};
```

Description
Close a UDP endpoint.

After sending this message, the object can no longer send or receive data. The endpoint will reply when the connection has been fully closed.

UDPEndpointCloseMsg is inherited from UDPEndpointBaseMsg.

Parameters
module Destination module reference

Returned value
See “UDP error” for a description of the possible UDP error codes.

See also
`UDPEndpointBaseMsg`

9 DNS reference

This chapter describes the DNS messages that an object sends to the IPStack to request DNS services. All messages are inherited from DNSEndpointBaseMsg.

DNS errors

Table 19 describes the errors that can be returned by any of the messages in this chapter. The possible error values are defined in the DNSEndpointError enumerated type.

When your object requests a network operation and receives a reply from its DNS endpoint, it should examine the request's error field (for example, DNSEndpointSetServerAddressesMsg.error).

Table 19 DNSEndpointError

Error value	Description
DNS_BUFFER_INVALID	Invalid use of a shared buffer.
DNS_CONNECTION_BUSY	The connection is busy and the requested operation cannot be completed. The DNS endpoint to which you sent the message may already be processing another request.
DNS_CONNECTION_CLOSED	The connection has been closed.
DNS_FAIL	The operation failed (no more information is available).
DNS_HOST_NOT_FOUND	No bind information could be found for the provided host name.
DNS_INDEX_INVALID	The specified index does not exist.
DNS_NO_DATA	No data record exists for the requested type.
DNS_NO_RECOVERY	A non-recoverable error occurred.
DNS_OPERATION_INVALID	The requested operation is not allowed in the current state.
DNS_OPERATION_UNKNOWN	The requested operation was not recognized by the DNS endpoint.
DNS_SUCCESS	Operation succeeded.
DNS_TRY_AGAIN	The requested host was not found, or the server request failed.

DNSEndpointBaseMsg

Definition

```
struct DNSEndpointBaseMsg: public antEnvMsg
  DNSEndpointError error;
};
```

Description
Defines the base message for all DNS network operation requests.

Requests for specific network services are sent in messages inherited from DNSEndpointBaseMsg, and are described later in this chapter.

Parameters
error (out) Returns the result of the operation.

See also
DNSEndpointSetServerAddressesMsg,
DNSEndpointGetServerAddressesMsg,
DNSEndpointSetDefaultDomainNameMsg,
DNSEndpointGetDefaultDomainNameMsg,
DNSEndpointGetHostByNameMsg, DNSEndpointGetHostByAddrMsg,
DNSEndpointGetAddressMsg, DNSEndpointGetAliasMsg,
DNSEndpointCloseMsg

DNSEndpointSetServerAddressesMsg

Definition

```
DNSEndpointSetServerAddressesMsg(
  int nscount,
  IPAddress addrList[MAXNS]
);
```

Description
Register a list of DNS servers that the object will use for resolving domain names and IP addresses.

The list identifies the DNS servers by their IP addresses. After this list is registered, queries will be sent to the servers in the order they appear in the list.

DNSEndpointSetServerAddressesMsg is inherited from DNSEndpointBaseMsg.

Parameters

nscount (in) The number of IP addresses to register.

addrList[MAXNS] (in) The list of IP addresses to register.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetServerAddressesMsg`, `antSharedBuffer`

DNSEndpointGetServerAddressesMsg

Definition

```
DNSEndpointGetServerAddressesMsg(
  int nscount,
  IPAddress addrList [MAXNS]
);
```

Description
Get a list of the DNS servers used by the object for resolving domain names and IP addresses.

The list identifies the DNS servers by their IP addresses. The addresses must have been set previously by DNSEndpointSetServerAddressesMsg.

DNSEndpointGetServerAddressesMsg is inherited from DNSEndpointBaseMsg.

Parameters

nscount	(out) The number of registered IP addresses.
addrList[MAXNS]	(out) The list of IP addresses.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointSetServerAddressesMsg`, `antSharedBuffer`

DNSEndpointSetDefaultDomainNameMsg

Definition

```
DNSEndpointSetDefaultDomainNameMsg(
  char name[MAXDNAME]
);
```

Description
Sets the default domain name for an object.

After the default domain name is registered, the name will automatically be added to all host names that are not fully qualified.

DNSEndpointSetDefaultDomainNameMsg is inherited from DNSEndpointBaseMsg.

Parameters
name[MAXDNAME] (in) The default domain name. This name must be null-terminated.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetDefaultDomainNameMsg`, `antSharedBuffer`

DNSEndpointGetDefaultDomainNameMsg

Definition

```
DNSEndpointGetDefaultDomainNameMsg(
  char name[MAXDNAME]
);
```

Description
Gets the default domain name for an object.

The domain name must have been set previously by DNSEndpointSetDefaultDomainNameMsg.

DNSEndpointGetDefaultDomainNameMsg is inherited from DNSEndpointBaseMsg.

Parameters
name[MAXDNAME] (out) The default domain name. This name is null-terminated.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointSetDefaultDomainNameMsg`, `antSharedBuffer`

DNSEndpointGetHostByNameMsg

Definition

```
DNSEndpointGetHostByNameMsg(
  char name[MAXDNAME],
  IPAddress server_address,
  IPAddress host_address,
  int n_address,
  int n_alias
);
```

Description
Get a list of IP addresses and domain name aliases for the host specified by domain name.
This list, called a host entry, will contain:

- The address of the DNS server that returned the host entry.
- The first IP address in the list (the official address), and a count of the total number of IP addresses for the host.
- The first domain name in the list (the official domain name), and a count of the total number of domain name aliases for the host.

If you want to get a different IP address or one of the domain name aliases, you must request them specifically. For details, see DNSEndpointGetAddressMsg and DNSEndpointGetAliasMsg.

DNSEndpointGetHostByNameMsg is inherited from DNSEndpointBaseMsg.

Parameters

name[MAXDNAME]	(in/out) The domain name of the host for which you want to get an entry. The domain name must be null-terminated. If you specify a domain name alias, this parameter will return the official domain name of the host.
server_address	(out) The IP address of the DNS server that sent the host entry.
host_address	(out) The IP address of the host. If the host has more than one IP address, this parameter returns the first IP address.
n_address	(out) The number of IP addresses for the host.
n_alias	(out) The number of domain name aliases for the host.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetHostByAddrMsg`, `DNSEndpointGetAddressMsg`, `DNSEndpointGetAliasMsg`, `antSharedBuffer`

DNSEndpointGetHostByAddrMsg

Definition

```
DNSEndpointGetHostByAddrMsg(
  IPAddress host_address,
  IPAddress server_address,
  char name[MAXDNAME],
  int n_address,
  int n_alias
);
```

Description
Get a list of IP addresses and domain name aliases for the host specified by IP address.

This list, called a host entry, will contain:

- The address of the DNS server that returned the host entry.
- The first IP address in the list (the official address), and a count of the total number of IP addresses for the host.
- The first domain name in the list (the official domain name), and a count of the total number of domain name aliases for the host.

If you want to get a different IP address or one of the domain name aliases, you must request them specifically. For details, see DNSEndpointGetAddressMsg and DNSEndpointGetAliasMsg.

DNSEndpointGetHostByAddrMsg is inherited from DNSEndpointBaseMsg.

Parameters

host_address	(in/out) The IP address of the host. This parameter returns the first IP address of the host, even if you specify a different IP address in the original message.
server_address	(out) The IP address of the DNS server that sent the host entry.
name[MAXDNAME]	(out) The official domain name of the host. The domain name is null-terminated.
n_address	(out) The number of IP addresses for the host.
n_alias	(out) The number of domain name aliases for the host.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetHostByNameMsg`, `DNSEndpointGetAddressMsg`, `DNSEndpointGetAliasMsg`, `antSharedBuffer`

DNSEndpointGetAddressMsg

Definition

```
DNSEndpointGetAddressMsg(
  int index,
  IPAddress address
);
```

Description
Get a registered IP address for the specified host.

Notes
Before you perform this operation, you must have already received the host entry. For details, see DNSEndpointGetHostByNameMsg or DNSEndpointGetHostByAddrMsg.

DNSEndpointGetAddressMsg is inherited from DNSEndpointBaseMsg.

Parameters

index (in) The index of the IP address that you want to get.

This index must be less than the value of n_address, returned in DNSEndpointGetHostByNameMsg or DNSEndpointGetHostByAddrMsg when the host entry was received earlier. A value of 0 returns the first IP address in the host entry list.

address (out) The IP address at location index in the host entry.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetHostByNameMsg`, `DNSEndpointGetHostByAddrMsg`, `DNSEndpointGetAliasMsg`, `antSharedBuffer`

DNSEndpointGetAliasMsg

Definition

```
DNSEndpointGetAliasMsg(
  int index,
  char name[MAXDNAME]
);
```

Description
Get a domain name alias for the specified host.

Notes
Before you perform this operation, you must have already received the host entry. For details, see DNSEndpointGetHostByNameMsg or DNSEndpointGetHostByAddrMsg.

DNSEndpointGetAliasMsg is inherited from DNSEndpointBaseMsg.

Parameters

index (in) The index of the domain name alias that you want to get.

This index must be less than the value of n_alias, returned in DNSEndpointGetHostByNameMsg or DNSEndpointGetHostByAddrMsg when the host entry was received earlier. A value of 0 returns the official domain name of the host.

name[MAXDNAME] (out) The domain name alias at location index in the host entry.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`, `DNSEndpointGetHostByNameMsg`, `DNSEndpointGetHostByAddrMsg`, `DNSEndpointGetAddressMsg`, `antSharedBuffer`

DNSEndpointCloseMsg

Definition

```
DNSEndpointCloseMsg();
```

Description
Close a DNS endpoint.

After sending this message, the object can no longer send or receive data. The endpoint will reply when the connection has been fully closed.
DNSEndpointCloseMsg is inherited from DNSEndpointBaseMsg.

Returned value
See “DNS error” for a description of the possible DNS error codes.

See also
`DNSEndpointBaseMsg`

10 IP reference

This chapter describes the IP messages that an object sends to the IPStack to request network services. All messages are inherited from IPEndpointBaseMsg.

IP errors

Table 20 describes the errors that can be returned by any of the messages in this chapter. The possible error values are defined in the IPEndpointError enumerated type.

When your object requests a network operation and receives a reply from its IP endpoint, it should examine the request's error field (for example, IPEndpointBindMsg.error).

Table 20 IPEndpointError

Error value	Description
IP_BUFFER_INVALID	Invalid use of a shared buffer.
IP_CONNECTION_BUSY	The connection is busy and the requested operation cannot be completed. The IP endpoint to which you sent the message may already be processing another request.
IP_CONNECTION_CLOSED	The connection has been closed.
IP_FAIL	The operation failed (no more information is available).
IP_INVALID_PROTOCOL	The specified protocol identifier is not valid.
IP_OPERATION_INVALID	The requested operation is not allowed in the current state.
IP_OPERATION_UNKNOWN	The requested operation was not recognized by the IP endpoint.
IP_PACKETSIZE	The specified packet size is incorrect.
IP_SUCCESS	Operation succeeded.

IP packet types

A number of different packets can be sent and received by an IP endpoint, as described in Table 21.

Notes
Normal IP packets are type IP_DATA. The other types are for ICMP packets.

Table 21 IP packet types

Packet type	Description
IP_DATA	Normal IP data.
IP_HOST_UNREACHABLE	(ICMP packet) The host cannot be reached.
IP_MESSAGE_TOO_LONG	(ICMP packet) The message was too long or could not be fragmented somewhere during the transmission.
IP_NETWORK_UNREACHABLE	(ICMP packet) The network cannot be reached.
IP_PORT_UNREACHABLE	(ICMP packet) The requested port could not be reached.
IP_PROTOCOL_UNREACHABLE	(ICMP packet) The requested protocol is not available on the target host.
IP_TIME_EXCEEDED	(ICMP packet) Packet reassembly timed out.
IP_TTL_EXCEEDED	(ICMP packet) TTL of packet has expired during transit.

IPEndpointBaseMsg

Member

```
IPEndpointBaseMsg(
IPEndpointError error
);
```

Description
Defines the base message for all IP network operation requests.

Requests for specific network services are sent in messages inherited from IPEndpointBaseMsg, and are described later in this chapter.

Parameters
error (out) Returns the result of the operation.

Returned value
See “IP error“ for a description of the possible IP error codes.

See also
`IPEndpointBindMsg`, `IPEndpointSendMsg`, `IPEndpointReceiveMsg`, `IPEndpointCloseMsg`

IPEndpointBindMsg

Member

```
IPEndpointBindMsg(
  Protocol protocol
);
```

Description
Bind an endpoint to a particular protocol.

All packets sent and received by the object over this endpoint will be identified as originating from the specified protocol.

The endpoint will reply to this message when the protocol has been bound and IP packets can be sent and received.

IPEndpointBindMsg is inherited from IPEndpointBaseMsg.

Notes
It is not possible to bind an endpoint to the TCP or UDP protocols, which have already been bound in the protocol stack. If attempted, the error IP_INVALID_PROTOCOL will be returned.

However, it is possible to bind an endpoint to ICMP. ICMP will process all packets that it recognizes, as usual, but will forward all unidentified packets to the endpoint that you have bound.

Parameters

protocol (in) The protocol to bind to the endpoint. Each protocol is identified by an integer with a value less than 256.

Returned value
See “IP error” for a description of the possible IP error codes.

See also
`IPEndpointBaseMsg`

IPEndpointSendMsg

Member

```
IPEndpointSendMsg(
  IPPacketType type,
  byte* buffer,
  int size
);
```

Description
Send an IP packet.

Any data that your object sends to the protocol stack must be stored in a shared memory buffer, defined by the antSharedBuffer structure. The IP endpoint replies to this message when the data has been copied from the shared buffer into the IPStack internal memory buffers.

IPEndpointSendMsg is inherited from IPEndpointBaseMsg.

Parameters

type (in) The type of packet being sent.
Normal IP packets are of type IP_DATA. See “Ip Packet types” for descriptions of all available packet types.

buffer (in) The location (in a shared buffer) where the packet being sent is stored.

size (in) The size of the packet being sent, in bytes.

Returned value
See “IP error” for a description of the possible IP error codes.

See also
`IPEndpointBaseMsg, antSharedBuffer`

IPEndpointReceiveMsg

Member

```
UDPEndpointReceiveMsg(
  IPPacketType type,
  byte* buffer,
  int size
);
```

Description
Receive an IP packet.
Any data that your object receives from the protocol stack is stored in a shared memory buffer, defined by the antSharedBuffer structure. The IP endpoint replies to this message when the data has been copied into the shared buffer. The size parameter returns the actual number of bytes received. If the received packet is too big for the shared buffer, only part of the packet will be stored in the buffer. The error IP_PACKETSIZE error will be returned.

IPEndpointReceiveMsg is inherited from IPEndpointBaseMsg.

Parameters

type (out) The type of packet received.
Normal IP packets are of type IP_DATA. See “IP packet types” for descriptions of all available packet types.

buffer (in) The location (in a shared buffer) where the packet being received should be stored.

size (in/out) Specifies the maximum number of bytes to receive. When the receive operation has completed and the endpoint has replied to the object, this parameter returns the actual number of bytes that were received. If the packet being received is too big for the shared buffer, only part of the packet will be stored in the buffer. The error IP_PACKETSIZE error will be returned.

Returned value
See “IP error” for a description of the possible IP error codes.

See also
`IPEndpointBaseMsg, antSharedBuffer`

IPEndpointCloseMsg

Member

```
IPEndpointCloseMsg();
```

Description
Close the IP connection.
After sending this message, the object can no longer send or receive data. The endpoint will reply when the connection has been fully closed.

IPEndpointCloseMsg is inherited from IPEndpointBaseMsg.

Returned value
See “IP error” for a description of the possible IP error codes.

See also
`IPEndpointBaseMsg`

Glossary

ANT environment
The ANT environment is the object in the OPEN-R system layer that offers networking services to objects. It communicates through normal message passing. It communicates directly with OPEN-R device drivers that exchange data over the physical network.

datagram
A term for packet in the UDP protocol.

Domain Name System (DNS)
A protocol that runs on top of the UDP layer in the IPv4 protocol stack. It offers services for setting, getting, and translating Internet domain names and IP addresses.

endpoint
A construct in the ANT environment that communicates with objects. Endpoints communicate with OPEN-R objects via message passing.

Objects have one endpoint for each open network connection. Endpoints are created dynamically at run time when the object requires a new network connection. Objects send requests for new endpoints to endpoint factories, which create the endpoints.

ephemeral port number
See port number.

Internet Protocol (IP)
A network protocol that is responsible for transmitting packets over the network. In the IPv4 protocol stack, IP is the bottom layer. Typically, objects do not communicate directly with IP. Instead, they open connections with layers on top of IP, such as TCP or UDP. However, some objects may be able to use the IP layer directly. For example, if you want to add new protocols without programming in the ANT environment, you can write the protocols as objects that communicate directly with the IP layer.

Internet Protocol version 4 (IPv4) protocol stack
A protocol stack in the ANT environment that offers the following network protocols: TCP, UDP, DNS, and IP.

packet
A unit of data that is sent over a physical network.

port number
A number that identifies a specific software process on a host or server. Some processes, such as FTP, are assigned permanent port numbers. These are called well-known port numbers. Some port numbers are assigned to processes temporarily for the duration of a network connection. These are called ephemeral port numbers.

SDU (service data unit)
The basic data container in the ANT environment. An SDU is a pointer to a series of data cells in an SDU pool. SDUs carry the data that is being sent over the network—called protocol data units (PDUs) in the OSI standard.

SDU pool
A memory buffer inside the ANT environment, which stores the data that is processed by the protocol stack. The ANT environment allocates service data units (SDUs) inside SDU pools.

shared memory buffer

A structure that maps a shared memory area into the address spaces of your object and the protocol stack. When your object exchanges data with the protocol stack, the data is identified by a pointer to the shared buffer and an offset in the buffer.

Transmission Control Protocol (TCP)
Runs on top of the IP protocol. It provides a connection-oriented, reliable, byte stream service.

User Datagram Protocol (UDP)
Runs on top of the IP protocol. It provides objects with an unreliable datagram delivery service.