



## Appendix B

### RTP Protocol Reference

---

#### B.1 RTP Protocol

##### B.1.1 Introduction

This document defines the **REF TEK** Protocol (**RTP**). **RTP** is designed to provide the application with a full-duplex, packet-oriented, reliable, transport over UDP network connections. The reader is assumed to have a working understanding of the TCP/IP protocol suite and networking concepts in general.

**RTP** is typically used in server-client fashion although this is by no means required. Typically there will be a server application running on a IP host somewhere on the network. Clients will attach themselves to the server to send and receive data. The client is typically an embedded system that attaches to the network through an asynchronous serial interface using Point-to-Point protocol however other interfaces will be implemented in the future. The server is typically an application program running on a host and accesses the network via UDP sockets provided by the IP stack on the local operating system.

The first implementation of **RTP** was on the RT422C Asynchronous Communications Card for the **REF TEK** 72A series Data Acquisition Systems (DAS) and the server application **RTPD**. **RTPD** and its associated software run on Windows 98, NT, 2000, XP, Linux, and Solaris (Intel and Sparc). Throughout this document we will cover some of the details of this implementation and use it to illustrate various design concepts.

## B.1.2 Design Goals

The following were the goals of the design for RTP:

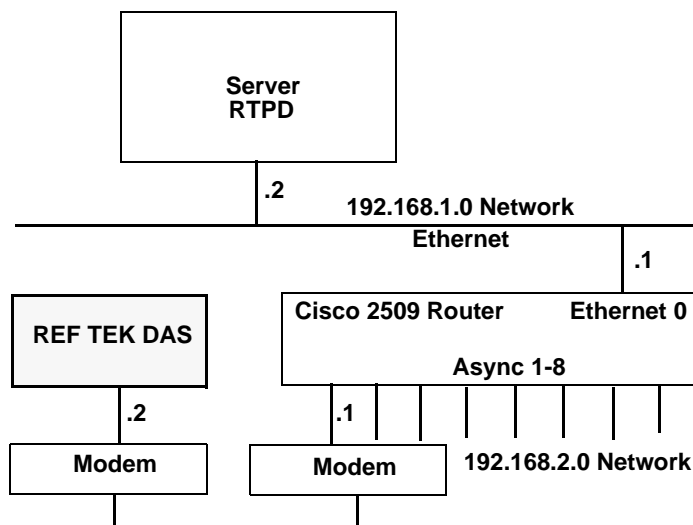
1. **Entirely platform independent**  
All data values are stored in network byte order. Can be Implemented on any hardware platform or OS that provides an IP protocol stack.
2. **Encapsulate the application data completely**  
Not have any dependency on the contents of any particular application data packet. That is to say that the protocol will be completely unaware of what it is transporting to the peer. The only requirement placed on the application is that the data packet be 1024 bytes or less in size.
3. **Self-contained and self-configuring at the client**  
The protocol stack must discover or be assigned all necessary parameters to operate from the network. No configuration information will be stored by embedded implementations nor will any higher level application configure or control it. The higher level application simply submits data packets to be sent as they are ready and will always be willing to receive data. However, the higher level application must respond to flow control from **RTP** to avoid loss of data.
4. **Both the server and client must initiate the connection on-demand**  
When the application has data to send the connection will be established if needed simply by submitting the data packet to be sent. If the connection is down, both must respond to link establishment by the peer at all times. There does not need to be an administratively opened or closed state, it is always administratively open.
5. **Recover from loss of connection without data loss**  
If a client is sending data to the server and the connection is lost momentarily, it will reestablish the link and resume sending data. No data may be lost or passed on out of order by the server. Momentary loss of connection will mean less than five minutes for purposes of the protocol.
6. **Deal with long, thin, pipes effectively**  
It must be capable of high utilization (>90%) of slow (9.6k), high latency (>1 second), connections such as VSAT links. We will use deep queues (16 slots) and adaptive retransmission time-out to achieve this goal.
7. **Small and relatively simple implementation**  
It must be suitable for embedding in dedicated communications hardware for **REF TEK** recording systems.

## 8. Function at the application layer

Uses the standard UDP Sockets API on the server system. All network traffic will be UDP datagram to/from the **REF TEK** port number, port 2543. This port is the *well-known* **REF TEK** port and is registered with the Internet Assigned Numbers Authority (IANA) for use with both UDP and TCP. UDP broadcasts will be used only during link establishment and will not be sent more frequently than one packet every ten seconds per client.

### B.1.3 Example Application

Throughout this document we will use the example system shown in Figure B - 1 to illustrate the details of **RTP**.

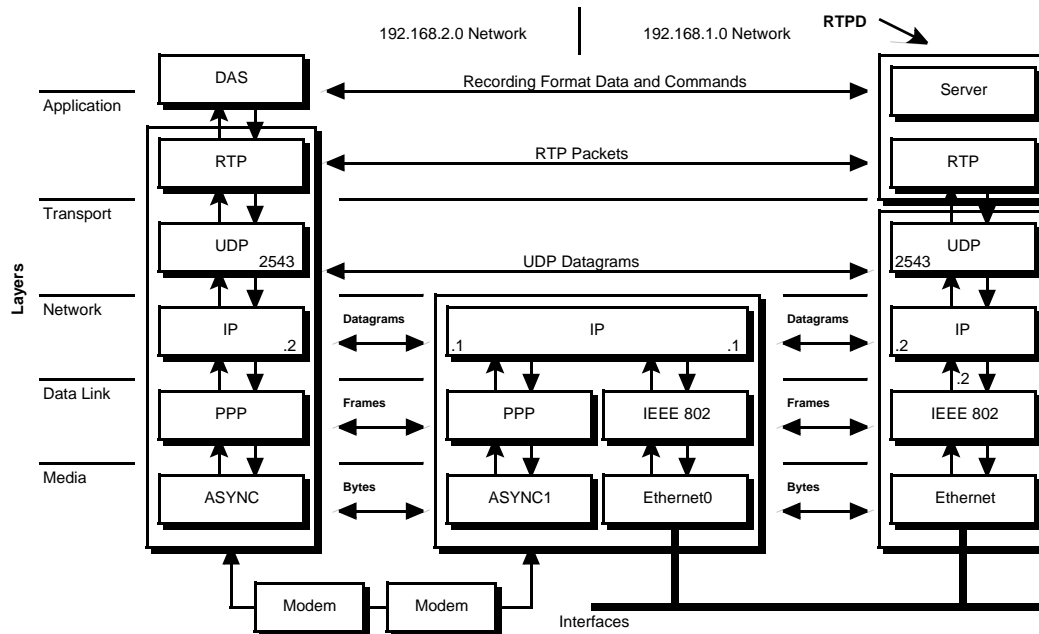


**Figure B - 1 The example network**

This setup consists of two class C networks interconnected by a router. The client is on one network (192.168.2) and connects to the router through an asynchronous serial interface using PPP. The server is on the other network (192.168.1). The router routes traffic between the two networks.

The DAS sends its recording format data to the server as it is acquired. The server sends command format packets to the DAS and receives responses as data packets.

Figure B - 1 on page B-87 and Figure B - 2 show a schematic representation of the example network.



**Figure B - 2 Layers vs. interfaces in the example network**

Data from the DAS flows down through the stack to the interface at the bottom and across the wire to the router. The router forwards the packet to its Ethernet interface and on to the server. It then flows up through the stack at the server to the application.



**Note:** In a router packets only come up the stack to the network layer and are routed. At the transport layer and above, the client views the connection from end-to-end and is unaware of any routing or interface issues. From the network layer down, only the next-hop host is visible and at the network layer all forwarding issues are dealt with.

## B.2 RTP Encapsulation

Application data is encapsulated by **RTP** for transport across the network and de-capsulated upon arrival. This is accomplished by prep ending an eight byte packet header, that contains the additional data required by the protocol to perform error-correction and various control functions, to the data packet.

The **RTP** header has the following form:

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	
8	Data...			

All values in any **RTP** packet are stored in *network byte order*. When transmitted on the network, bytes are sent from bit 0 to bit 7, multi-byte values are sent LSB first, MSB last. This is also known as *big-endian* byte order and is the same as specified for the entire TCP/IP protocol family.

The **RTP** packet is further encapsulated within a UDP datagram as it passes through the transport layer on its way down the protocol stack. All UDP encapsulated **RTP** packets will typically be sent to and received from the well-known **REF TEK** port number 2543. However, the server may serve clients on ephemeral ports if so desired.

### B.2.1 RTP Protocol Field

The Protocol field is used to identify the contents of this UDP datagram as a **RTP** packet. The **RTP** protocol number is 0x4023. An early version of **RTP** was implemented at the network layer directly on top of PPP at the data link layer for use over dedicated asynchronous serial links (RT422A and B). The 0x4023 protocol was registered as a PPP protocol number with IANA at that time. We have used the same value here simply because we already have this number and it will serve this purpose adequately.

## B.2.2 RTP Packet Codes

The Code field is an 8 bit unsigned integer that is used to identify the type of **RTP** packet. Up to 256 types of packets may be used by **RTP**.

**There are currently nine different packet types that fall into three classes:**

1. Data packets used to transport data over the connection to the peer.
2. Server Discovery packets used by a client to find a server.
3. Synchronization packets used to synchronize the sequence numbers used at each end of the connection for error-correction.

Table 1 on page 90 lists the types of packets currently defined for **RTP**.

**TABLE 1. Currently defined RTP Codes**

Code	Binary code	Name	Description
0x0	00000000	Data	Application data (payload).
0x01	00000001	DataAck	Data acknowledgement, data packet accepted by peer
0x04	00000100	Sync	Synchronize outbound sequence number.
0x05	00000101	SyncAck	Acknowledgement of sequence number.
0x06	00000110	USync	Unconditional synchronize.
0x07	00000111	USyncAck	Acknowledgement of unconditional synchronize.
0x08	00001000	SvrInquiry	Server discovery inquiry.
0x09	00001001	InquireAc	Server acknowledgement.
0x0B	00001011	InquireNak	Server negative acknowledgement.

All codes not listed in Table 1 on page 90 are reserved for future use. Note that as defined, bit 0 of the code field may be interpreted as the “ack bit”, bit 1 as the “unconditional bit”, bit 2 as the “sync bit”, and bit 3 as the “discovery bit”.

The three classes of packets can be easily distinguished by checking bits 2 and 3 of the code field. If bit 2 is set, this is a synchronization packet, bit 3 indicates a server discovery packet, otherwise it is data.

### B.2.3 RTP Sequence Numbers

The Sequence Number field is an 8 bit unsigned integer that is used to correct transmission errors. The value of this field ranges from 0 to 255.

As each data packet is sent, the sequence number increments. The receiving end generates an acknowledgment packet for each packet that is accepted with the same sequence number and sends it back to the sender. The sender then knows that packet has been successfully transported to the peer.

The sequence number is used by the receiver to reorder packets, detect duplicate packets, and detect old packets. More details about data transfer can be found in **RTP** Data Transfer below.

### B.2.4 RTP Unit ID Field

The Unit ID field is a 16 bit unsigned integer that identifies the application at the peer. In the case of the RT422C card, this is the DAS unit ID number. However, the purpose of this number is to identify the peer at the opposite end of the connection, which is not necessarily a DAS. This field should be thought of as a peer, or connection, number.

### B.2.5 RTP Length Field

The length field is a 16 bit unsigned integer number of bytes in the **RTP** packet including the **RTP** header. This means that if there are 0 data bytes, the length field will be set to 8.

## B.3 RTP Operation

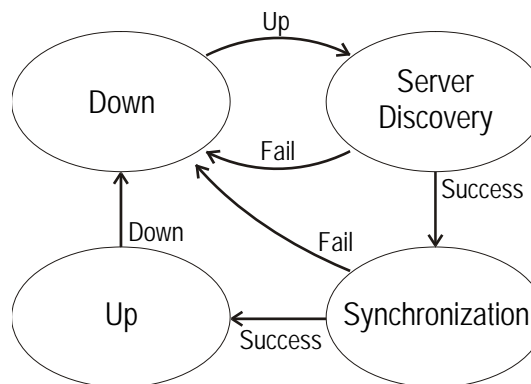
In order for a client to move data across the network to a server, it must connect to the network, discover the address of the server, and then synchronize sequence numbers.

**In order to accomplish this, RTP goes through several distinct phases.**

1. **Down**, the link is not available for data transfer.
2. **Server Discovery**, the client is looking for the server.
3. **Synchronization**, client and server are synchronizing sequence numbers.
4. **Up**, the link is available for data transfer.
5. If a server wishes to **connect** to a client, it must **synchronize** but it will not need to discover the client's address, so the discovery phase is simply skipped.

### B.3.1 Phase Diagram

Figure B - 3 on page B-92 shows the relationships between these phases.



**Figure B - 3 RTP connection phases**

### B.3.2 Down

While **RTP** is in the Down phase, the connection is not available for data transfer. The connection begins and ends in this phase.

One of two events signals that the connection should be established. If the application submits a data packet for transfer, the lower layers are opened if necessary, and **RTP** proceeds to the Server Discovery phase. Clients will also respond an incoming connection, that is, the transport layer signals an Up event. Servers are generally always connected to the network and respond to Server Inquiry packets to establish inbound connections.



---

### B.3.3 Server Discovery

This is the only aspect of **RTP** that follows a client-server model and allows a client to *discover* its server. This mechanism meets the self-configuration requirements put forth in the design goals above.

Clients must discover the IP address and port number (the *end-point*) that the server will use to service its connection. Once connected to the network, the client periodically sends Server Inquiry packets containing the endpoint as subnet-directed broadcasts to the well-known **REF TEK** port. If the client has never been connected, the endpoint is 0.0.0.0:2543, if it has, it is the endpoint last used. The server listens for these broadcasts and responds with a InquireAck if the endpoint is correct or a InquireNak that contains the endpoint that it desires.

Once the client receives an InquireAck from the server, it proceeds to the Synchronize phase.

Servers simply skip this entire phase and proceed directly to the Synchronize phase. The server must always accept and respond to Discovery packets that it receives.

### B.3.4 Synchronize

In order to perform error-correction, each end of the connection must notify the other of the sequence number of the next Data packet that it will send and receive acknowledgement. This can be accomplished by a finite-state automaton and is covered in detail below.

Note that there are two synchronize packets defined, Sync, and USync. The normal Sync, sometimes referred to as a *warm* Sync indicates that there has been a previous connection and we are attempting to resume that connection. The unconditional or USync, sometimes referred to as a *cold* Sync indicates that the sender has never been connected and that the receiver should not attempt to resume any connection.

Once an **Ack**, either **SyncAck**, or **USyncAck**, has been both sent and received, **RTP** proceeds to the Up phase. While in the Synchronize phase, no Data class packets are allowed to be transferred. Any Data class packets received are silently discarded.

### B.3.5 Up

While **RTP** is in the Up phase, data may flow across the connection. All types of **RTP** packets are allowed and data moves across the connection error-free.

## B.4 RTP Server Discovery

Server Discovery is a mechanism used by **RTP** clients to discover the IP address and UDP port number that the server will use to service its connection. This mechanism is the only aspect of **RTP** that employs the client-server model.

**RTP** servers must listen on UDP port 2543, the well-known **REF TEK** port, for **RTP** Server Inquiry packets from clients. Clients send these packets as subnet-directed broadcasts. These packets carry as data the endpoint that will be used by the **RTP** server to service the connection.

### B.4.1 The Discovery Process

After a client has successfully attached to the network (an Up from the transport layer), it begins periodically (typically every ten seconds) broadcast a Server Inquiry packet to the **REF TEK** port. This will continue until successful discovery is achieved or the network connection goes down.

When a server receives a Server Inquiry packet, it examines the contents of the packet, specifically the Unit ID, and endpoint, and either reactivates a previous connection for the Unit or establishes a new connection for the Unit. If the connection is reestablished, the server can simply send an InquireAck to the peer. If a new connection is being activated, the server sends an InquireNak to the peer that contains the endpoint that the client must use to communicate with the server. Note that the server should unicast the Ack packet to the source endpoint of the Server Inquiry packet.

When a client receives an InquireAck from a server, it simply uses the endpoint within the packet and proceeds to the Synchronize phase. When a client receives an InquireNak from a server, it broadcasts a new Server Inquiry packet containing the new endpoint and the process continues.

A client will not leave the Discovery phase until it receives an InquireAck or the transport layer signals a down event. This mechanism ensures that the server and the client agree on the server's endpoint. If a packet is lost during this process it will still result in successful discovery.

---

## B.4.2 Network Issues

Because the client broadcasts **Server** Inquiry packets, there can be problems in your network caused by routers not forwarding these packets to other networks. It is required that the server be visible to the client through subnet-directed broadcasts in order for the server discovery process to succeed.

If the server and client both reside on the same network, this is not an issue. However, in the real world, this may not be the case. In the case of our example system (see Figure B - 1 on page B-87), the server is on the other side of a router and measures must be taken to insure that the discovery process can succeed.



**Note: the following aspects of the Server Inquiry packet:**

- [1] They are always a subnet-directed broadcast. In our example, a class C network, they are sent to 192.168.2.255.
- [2] They are always a UDP packet broadcast to the well-known Ref Tek port (2543).

Given this information, routers generally can be configured to allow limited forwarding of these broadcasts. In our example, the router is configured to forward only UDP packets destined for port 2543 to the subnet-directed broadcast address (192.168.1.255) of the other network and this solves the problem. The responses from the server are unicast back to the client and need no special attention.

Most routers provide this ability because various protocols, such as BOOTP, must provide this same type of functionality by employing UDP broadcasts.

See the section below for the configuration of a Cisco 25xx router that handles the issues in the example network.

### B.4.3 Discovery Class Packets

The three packets used in the server discovery process in detail are:

1. **ServerInquiry** - packet is sent as a subnet-directed UDP broadcast to the well-known **REF TEK** port (2543). The endpoint of the server is carried in the packet as six bytes of data. It is used by the client to query the server as to the server endpoint to use for the **RTP** connection.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	
8	Server IP Address			
12	Port #			

Label	Description
<b>Code:</b>	0x08 (SvrInquiry)
<b>Unit ID:</b>	The unit ID of the client. The server will use this to identify the client.
<b>Length:</b>	14
<b>Server IP Address:</b>	32 bit unsigned integer IP address of the server. If unknown, 0.0.0.0, otherwise, the address last used for the RTP connection.
<b>Port #:</b>	16 bit unsigned port number of server. If unknown, 2543 (0x09EF), otherwise, the UDP port last used for the RTP connection.



**Note:** The sequence numbers used for Discovery and Synchronize class packets should be distinct from that used for Data class packets. During the discovery phase, the sequence number is simply used to match inquiries with responds. The client must increment the sequence number each time it sends a SvrInquiry packet.

The server must use the same sequence number in its response, either InquireAck, or InquireNak.

2. InquireAck - packet is a positive server response to a SvrInquiry packet from a client. It is used to confirm the server endpoint to be used by the client for the **RTP** connection.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	
8	Server IP Address			
12	Port #			

Label	Description
<b>Code:</b>	0x09 (InquireAck)
<b>Unit ID:</b>	The unit ID of the client. The server will use this to identify the client.
<b>Length:</b>	14
<b>Server IP Address:</b>	32 bit unsigned integer IP address of the server.
<b>Port #:</b>	16 bit unsigned port number of server.

The sequence number is the sequence number of the inquiry packet for which this packet is the response. Together, the Server IP Address and Port number make up the server endpoint to be used by the client for the **RTP** connection.

3. InquireNak - packet is a negative server response to a SvrInquiry packet from a client. It is used to communicate to the client the server endpoint that must be used for the **RTP** connection.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	
8	Server IP Address			
12	Port #			

Label	Description
<b>Code:</b>	0x0B (InquireNak)
<b>Unit ID:</b>	The unit ID of the client. The server will use this to identify the client.
<b>Length:</b>	14
<b>Server IP Address:</b>	32 bit unsigned integer IP address of the server.
<b>Port #:</b>	16 bit unsigned port number of server.

The sequence number is the sequence number of the inquiry packet for which this packet is the response. Together, the Server IP Address and Port number make up the server endpoint to be used by the client for the **RTP** connection.

## B.5 RTP Link Synchronization

RTP link synchronization is a process whereby an **RTP** implementation notifies its peer of its outbound sequence number. Each end of an **RTP** connection maintains two sequence numbers, they are the inbound and outbound sequence numbers. Each end is only responsible for synchronizing the outbound sequence number with the peer.

The process is complete when each end has both sent and received an acknowledgment packet from the peer. A finite-state-automaton (**FSA**) is provided that will accomplish this process.

During the synchronization process, the outbound sequence number is the number of the first data packet that will be sent to the peer. The inbound sequence number is the number of the next packet that will be forwarded to the application by **RTP**.

### B.5.1 Synchronization Class Packets

There are four synchronization class packets. They are presented in detail below. None of these packets carry data, that is, there is no additional data beyond the **RTP** header.

The four synchronization packets are:

1. **Sync** - packet is used to inform the peer of the sequence number that will be used on the next data packet that it will receive. This is a "normal" Sync, this means there was a previous connection to the peer and that the peer should check to see if the sequence number falls within its inbound sequence space. If it does, the connection is resumed, otherwise the action is the same as for USync (see below).

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	

Label	Description
<b>Code:</b>	0x04 (Sync)
<b>Sequence:</b>	Local outbound sequence number.
<b>Unit ID:</b>	The unit ID of local <b>RTP</b> .
<b>Length:</b>	8

2. **SyncAck** - packet is used to acknowledge a **Sync** packet received from the peer. The packet can be created by simply copying the received **Sync**, changing its code to **SyncAck**, and reflecting it back to the peer.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	

Label	Description
<b>Code:</b>	0x05 (SyncAck)
<b>Sequence:</b>	Peer's outbound sequence number.
<b>Unit ID:</b>	The unit ID of the peer <b>RTP</b> .
<b>Length:</b>	8

3. **USync** - or unconditional synchronize packet, is used to inform the peer that it has never been successfully connected to the peer and has no out-of-order data in its outbound queue. The peer must flush any pending out-of-order data that might be in its inbound queue and set its inbound sequence number.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	

Label	Description
<b>Code:</b>	0x06 (USync)
<b>Sequence:</b>	Local outbound sequence number.
<b>Unit ID:</b>	The unit ID of local <b>RTP</b> .
<b>Length:</b>	8

4. **USyncAck** - packet is used to acknowledge a **USync** packet received from the peer. The packet can be created by simply copying the received **USync**, changing its code to **SyncAck**, and reflecting it back to the peer.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	

Label	Description
<b>Code:</b>	0x07 (USyncAck)
<b>Sequence:</b>	Peer's outbound sequence number.
<b>Unit ID:</b>	The unit ID of the peer <b>RTP</b> .
<b>Length:</b>	8

## B.5.2 The Link Synchronization Automaton

Here is presented a finite-state-automaton (**FSA**) that will drive the link synchronization process.

The **FSA** is defined by events, actions, and state transitions. Events include: reception of external signals such as open and close commands and signals from lower layers, reception of **RTP** synchronization class packets, and the expiration of the restart timer. Actions include: communicating with the lower layers and transmitting packets to the peer.

The following table summarizes the events handled and actions taken by the automaton.

Actions:		Events:	
tls	This layer start.	Up	Lower layer is up.
tlf	This layer finished.	Down	Lower layer is down.
tlu	This layer up.	Open	Open the connection.
tld	This layer down.	Close	Close the connection.
irc	Initialize restart counter.	TO+	Time-out with restart counter > 0.
ssp	Send synchronize packet.	TO-	Time-out with restart counter = 0.
sap	Send sync acknowledge packet.	RSP	Received synchronize packet.
		RAP	Received sync acknowledge packet.

TABLE 2. FSA events and actions



### B.5.3 State Transition Table

The complete state transition table follows. States are indicated horizontally and events are read vertically. State transitions and actions are represented in the form *action/next-state*. Multiple actions are separated by commas and may continue on the next line. The dash indicates an illegal action.

Events/ state	0-Closed	1-Stopped	2-Sync-sent	3-Ack-rcvd	4-Ack-sent	5-Opened
Up	irc, ssp/2	irc, ssp/2	–	–	–	–
Down	1	1	1	1	1	tld/1
Open	tls/1	tls/1	2	3	4	tld, irc, ssp/2
Close	tlf/0	tlf/0	tlf/0	tlf/0	tlf/0	tld, tlf/0
TO+	–	–	ssp/2	ssp/2	ssp/4	–
TO -	–	–	tld, tlf/1	tld, tlf/1	tld, tlf/1	–
RSP	–	irc, ssp, sap/4	sap/4	sap, tlu/5	sap/4	tld, ssp, sap/4
RAP	–	irc, ssp/2	irc/3	ssp/2	irc, tlu/5	tld, irc, ssp/2

TABLE 3. Complete FSA state transition table

The states in which the restart timer is running are identified by the presence of the TO events.

If the link has never been in the Opened state (it is “cold”), unconditional Syncs and Sync Acks (USync, USyncAck) are sent to the peer, otherwise regular Syncs and SyncAcks are sent.

The Closed state (0) differs from the Stopped state (1) only in that the link is dropped (tlf) before the transition to Closed. In either state an incoming connect brings this layer up. This allows the application to use the Open and Close events as initiate and drop the link commands respectively. The automaton is in the Closed state only when the application commands it. It is in the stopped state due some external factor such as loss of carrier.

## B.5.4 States

The following is a brief description of each **FSA** state.

State	Description
Closed (0)	In this state, <b>RTP</b> has not been initialized and the transport layer is not up. No packets can be sent or received in this state. An <b>RTP</b> implementation should issue an open event as soon as possible at startup. The restart timer is not running.
Stopped (1)	In this state the transport layer is not up. The <b>FSA</b> will begin to synchronize the link when the transport layer signals an Up event. An Open event will cause the start action to initiate the connection. The restart timer is not running.
Sync-sent (2)	In this state a Sync (or USync) has been sent to the peer but a SyncAck (or USyncAck) has not been received. The restart timer is running and the Sync packet will be sent again upon its expiration.
Ack-rcvd (3)	In this state a Sync has been sent to and a SyncAck has been received from the peer. However, no Sync has been received and no Sync-Ack has been sent. The restart timer is running and the Sync packet will be sent again upon its expiration.
Ack-sent (4)	In this state a Sync and a SyncAck have been sent to the peer. No SyncAck has been received. The restart timer is running and the Sync will be sent again if it expires.
Opened (5)	In this state a SyncAck has been both sent to and received from the peer. The connection is now synchronized and ready to transfer data. The <b>RTP</b> connection is Up and the restart timer is not running.

## B.5.5 Events

FSA actions and state transitions are caused by events. The following is a brief summary of the events handled by the **FSA**.

Event	Description
Up	This event comes from the transport layer to indicate that the network will now accept traffic. On clients, this event is intercepted to initiate the server discovery process and upon successful completion is then sent to the FSA to cause the link to synchronize.
Down	This event comes from the transport layer to indicate that the connection to the network is down and unavailable for traffic. This causes the FSA to transition to the Stopped state.
Open	This event comes from the application and indicates that it desires to send data across the link. If the transport layer is not up, it causes action to initiate the connection to the network. If the FSA is already opened, this event causes re synchronization.
Close	This event comes from the application and indicates that the connection should be dropped. The connection to the network is broken and the FSA transitions to the Closed state.
Time-out TO+, TO-	<p>These events occur when the restart timer expires. The restart timer is used to time the response to Sync packets.</p> <p>The TO+ event indicates that the restart timer expired with the restart counter greater than zero. The restart is decremented and the packet is retransmitted.</p> <p>The TO- event indicates that the restart timer expired with the restart counter equal to zero. The link is recycled, that is, the connection to the network is dropped, then reestablished. Note that this causes server discovery to happen again before synchronization.</p>
Received Sync Packet (RSP)	<p>This event occurs when a Sync (or USync) packet is received from the peer. If the packet is a Sync, the sequence number within the packet is check against the inbound sequence space, if it is within the space, it is simply Ack'ed. If not, it is treated as a USync.</p> <p>If the packet is a USync, the inbound queue is flushed and the inbound sequence number is set to the sequence number within the USync packet. Because this informs the us that the peer is "cold", this forces the local system to be "cold" as well. The local outbound queue must be examined for non-contiguous packets.</p> <p>If any are found, they must be discarded and the outbound sequence number must be set to the oldest contiguous packet in the outbound queue and a USync must be transmitted.</p>
Received SyncAck packet (RAP)	This event occurs when a SyncAck (or USyncAck) is received from the peer. If the sequence number in the packet doesn't match that of the last transmitted Sync (or USync) the packet must be discarded.

## B.5.6 Actions

Actions are taken by the automaton as events occur. The following is a brief description of the various actions that may be taken by the FSA.

Action	Descriptions
Illegal Action (-)	This is an action that should never occur in a properly implemented automaton. This indicates an internal error that needs to be corrected.
This Layer Start (tls)	This action causes initiation of the connection to the network. Usually this simply causes an Open event to the transport layer which in turn opens the layers on down the stack.
This Layer Up (tlu)	This action occurs as the synchronization process is successfully completed and is used to perform whatever task is required at that point. This is the last action taken as the connection comes opened and may be used to notify the application the RTP connection is now ready for traffic.
This Layer Down (tld)	This action occurs when the connection is no longer ready to carry data traffic. It is the first action taken as <b>RTP</b> leaves the Up phase and can be used to notify the application that the connection is down.
This Layer Finished (tlf)	This action causes disconnection from the network. This is the last action taken before entering the Down phase.
Initialize Restart Counter (irc)	This action set the restart counter to the appropriate retry value. This is typically 10.
Send Sync Packet (ssp)	<p>This action is used to create a <b>Sync</b> or <b>USync</b> packet and send it to the peer.</p> <p>For a normal <b>Sync</b>, the sequence number of the packet is the number of the oldest packet in the outbound queue.</p> <p>For a <b>USync</b>, the outbound queue must be checked and any non-sequential packets discarded before the oldest packet sequence number is used.</p>
Send SyncAck Packet (sap)	This action is used to create a SyncAck or USyncAck packet and send it to the peer. The packet can be created by copying the received Sync or SyncAck, changing the code to an Ack (set bit 0), and reflect the packet to the peer.

## B.5.7 Counters and Timers

The **RTP** FSA makes use of one counter, the restart counter, and one timer, the restart timer. The restart counter is decremented each time that as **Sync** or **USync** packet is sent. Expiration of the restart timer causes the TO events that cause retransmission of these packets.

By default the restart timer should be set to six (6) seconds and the restart counter should be set to 10.

## B.6 RTP Data Transfer

The primary purpose of **RTP** is to transfer data. The requirements are, in order;

- Provide error-free transfer across the network.
- Efficient utilization of the available bandwidth.

The transport layer (**UDP**) provides a simple, packet oriented, best-effort service to **RTP**. UDP packets may arrive at the peer out of order or not at all.

In order to meet these requirements, **RTP** employs an acknowledgement mechanism, 16 slot queues to allow streaming transmission (up to 16 packets may be sent before receiving acknowledgement) on the outbound side, and reorder of packets on the inbound side. In addition to this queuing, an adaptive retransmission time-out scheme is used to insure that lost packets are retransmitted as soon as possible and helps to reduce the need for deep queues.

### B.6.1 Data Class Packets

Application data packets are encapsulated in **RTP** Data packets for transport across the network. The Data packet is complemented by the DataAck packet which is sent by the peer to indicate that it has accepted the Data packet.

#### Data

The Data packet is used to transport application data packets to the peer. Each packet is assigned a sequence number as they are received from the application and are delivered at the peer application in that same order.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	
8	Application Data...			

Label	Description
<b>Code:</b>	0x00 (Data)
<b>Sequence #:</b>	Senders outbound sequence number.
<b>Unit ID:</b>	The unit ID of the sending <b>RTP</b> .
<b>Length:</b>	8 + length of application data.
<b>Application Data:</b>	0 to 1024 bytes of payload data.

### DataAck

The DataAck packet is sent by the receiving **RTP** to indicate the acceptance of the associated Data packet. The sender of the Data packet takes reception of this DataAck packet to indicate that the packet has been successfully transported and can now be disposed of.

	0	1	2	3
0	Protocol (0x4023)		Code	Sequence #
4	Unit ID (0000-9999)		Length	

Label	Description
<b>Code:</b>	0x01 (DataAck)
<b>Sequence:</b>	Received Data packet sequence number.
<b>Unit ID:</b>	The unit ID of the <b>RTP</b> that is the source of the Data packet.
<b>Length:</b>	8

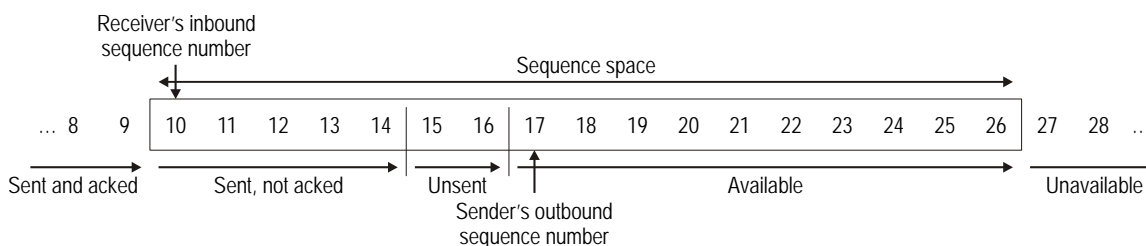


**Note:** An ack can be created by copying the RTP header of the Data packet, changing its code to DataAck and length to 8, and reflecting the packet to the peer.

## B.6.2 RTP Sequence Numbers

Every data packet sent across an **RTP** connection is assigned an 8 bit sequence number. This number is assigned as packets are accepted from the application by the sending **RTP** and reflect the order in which the application submitted them. **RTP** will deliver these packets in this same order to the application at the opposite end of the connection.

Because **RTP** is *full-duplex*, each end must maintain a set of sequence numbers, one for inbound and one for outbound packets. For each direction on the connection there is an available *sequence space*. This is the range of sequence numbers that are currently active in one direction.



**Figure B - 4 RTP sequence space**

The sender should not send packets with sequence numbers that are outside of the sequence space. The sender tracks the sequence space as starting at the oldest unacknowledged packet to that sequence number plus the depth of the queues (16).

The receiver tracks the sequence space from the inbound sequence number, that is the sequence number of the next packet that will be sent to the application, to that number plus the depth of the queue, again, 16. The receiver will accept, that is acknowledge, all packets up to and including the end of the sequence space. Packets with sequence numbers less than the inbound sequence number are old packets which have already been received and are discarded. Packets that already exist in the queue are duplicates and again are discarded. Packets greater than the end of the sequence space are in error and are not accepted, that is, they are not acknowledged and will be retransmitted by the sender. Properly implemented, **RTP** should never send packets that are greater than the end of the sequence space.

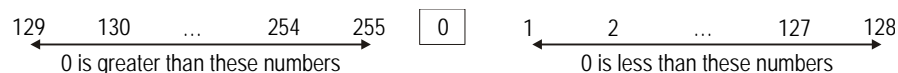
- Modular Arithmetic with Sequence Numbers - **RTP** must deal with the fact that the 8 bit integer sequence has a limited range of 0 to 255. Once  $2^8$  (256) packets have been sent, the sequence number will *wrap* from 255 back to 0. Given that the sequence space is limited to 16, this can easily be dealt with by employing modular arithmetic to compare sequence numbers.
- Sequence Number Comparisons - **RTP** sequence numbers are defined as an unsigned 8 bit integer. The following C language macros compare **RTP** sequence numbers:

```
typedef INT8 signed char;
```

```
#define SEQ_LT(a, b) ((INT8)((a) - (b)) < 0)
#define SEQ_LE(a, b) ((INT8)((a) - (b)) <= 0)
#define SEQ_GT(a, b) ((INT8)((a) - (b)) > 0)
#define SEQ_GE(a, b) ((INT8)((a) - (b)) >= 0)
```

When comparing two sequence numbers, we can simply subtract one from the other and interpret the result as a signed integer. This resulting relationship to zero is the relationship of the one sequence number (a) to the other (b).

This is somewhat counterintuitive. For example, if we compare 255 to 0, we find the 255 is less than zero.



**Figure B - 5 Sequence number comparison to zero**

Figure B - 6 shows a somewhat more intuitive way to visualize the circular nature of **RTP** sequence numbers.

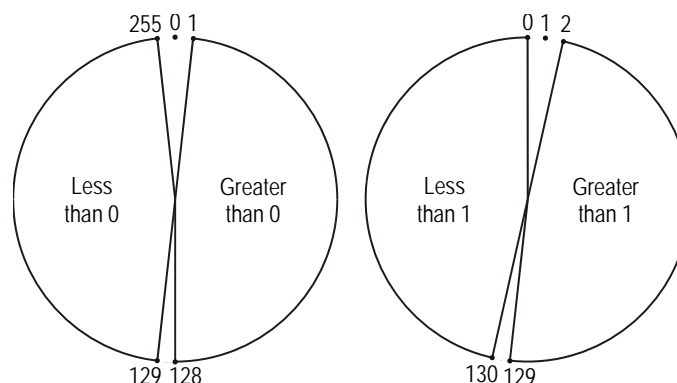


Figure B - 6 Comparison to 0 and 1

### B.6.3 RTP Outbound Processing

Application data packets submitted to **RTP** for transport across the network to the peer are encapsulated in **RTP** Data packets and added to the outbound queue. Each entry in the queue is called a slot and along with the **RTP** Data packet it has a counter and a timer associated with it. The counter is used to count the number of times that this packet has been sent and the timer is the elapsed time since it was last sent. There must be at least 16 slots available in the outbound queue.

There are two global values that are associated with outbound data, they are; the current outbound sequence number, and the current retransmission interval.

#### Enqueuing Outbound Data Packets

The queue is kept in order of sequence numbers and represents the currently occupied portion of the sequence space. The queue is typically implemented as a linked list of structures in memory. The head slot in the queue is the oldest, that is, the lesser sequence number, and the tail is the newest.

Before **RTP** enqueues an outbound packet, it must check that there is room in the sequence space for that packet. This is easily accomplished by comparing the sequence number of the head slot (the oldest unacknowledged packet) plus 16 with the current outbound sequence number. If the current outbound sequence number is less than the head number plus 16, then the packet may be enqueued, otherwise, the outbound queue is full and the application must wait.

New packets are assigned the current outbound sequence number and it is then incremented. The new packet is placed in the tail slot of the queue, its counter is set to 10, and its timer is initialized.



## Sending Outbound Data Packets

Packets are transmitted by examining the queue from head to tail and sending the first packet that either has never been sent (counter equal to 10) or has an elapsed time since last sent greater than the retransmission time-out. This insures that packets that need retransmission will get it in a timely manner.

As each slot of the queue is examined, several checks are made. First, if the counter on any packet is zero, there is a major problem with the connection to the network and it should be dropped and reestablished. Second, if the elapsed time since last sent is greater than the retransmit interval or the counter is equal to 10, the packet is to be sent. After the packet is sent, its counter is decremented, its timer restarted, and we repeat the procedure starting at the head of the queue, not the next slot.

If we reach the tail of the queue, no packets are ready to be sent so we simply repeat the procedure starting at the head slot again.

## Dequeuing Outbound Data Packets

As DataAck packets are received, the outbound queue is examined for the sequence number contained in the DataAck packet. If the corresponding Data packet is in the outbound queue, it is dequeued. The elapsed time since sent is the effective round-trip time to the server and may be used to implement adaptive retransmission.

## Adaptive Retransmission Time-out

Adaptive retransmit is a mechanism that keeps the retransmission interval set to a realistic value so that **RTP** does not stall or retransmit unnecessarily. The idea is to update the retransmission interval each time that a packet is dequeued from the outbound queue using a running average over some number of packets plus some time for overhead.

This is very helpful on connections with long round trip times. If the retransmission interval is too long, when a Data or DataAck packet is lost, **RTP** will send all packets in the outbound queue and then stall waiting for the missing DataAck to arrive before retransmitting the Data packet again. Conversely, if the interval is too short, it will retransmit Data packets before the DataAck can make the trip back and squander the bandwidth available on the connection.

Adaptive retransmit helps by insuring that Data packets needing retransmission are sent again quickly and insures that there are no more packets in the outbound queue at any given time than are absolutely necessary.

The following C language function will compute the retransmission interval based on the round trip time measured for the last packet transferred:

```
typedef signed long INT32;      /* 32 bit signed */
typedef unsigned long UINT32;  /* 32 bit unsigned */

#define RETRANS_MAX (10 * SECOND_MS)
#define RETRANS_MIN(500 * MSECOND_MS)
#define RETRANS_C 4
#define RETRANS_M 2

UINT32 ComputeRetransInterval( UINT32 round_trip, UINT32
    *retrans_interval )
{
    INT32 a, t, c;

    /* Compute average interval over last C packets by recursive
       approximation. The result is the minimum plus M times the
       average constrained to less than the maximum.

       round_trip = round trip time measured for last packet in ms.
       retrans_interval = pointer to the current retransmission
       interval in ms.

       return value = value of new retransmission interval.
    */

    t = (INT32)RETRANS_MIN + ((INT32)round_trip *
        (INT32)RETRANS_M);
    a = (INT32)*retrans_interval;
    c = (INT32)RETRANS_C;

    a = (a + ((t - a) / c));

    if (a > RETRANS_MAX)
        a = RETRANS_MAX;

    *retrans_interval = (UINT32)a;

    return(*retrans_interval);
}
```

If an error occurs and **RTP** retransmits a Data packet, it cannot be sure if the DataAck that it receives is for the last or the first packet sent. This means that measures must be taken to ensure that the retransmission interval is not allowed to decrease without limit. The code above will not allow it to decrease to less than 500 ms. However, a sudden change in the condition of the connection can still cause problems.

A simple solution is to check the counter on Data packets before they are dequeued. If the packet has been transmitted more than three times, it is likely that the interval needs to be increased.

Rather than recompute the interval, it should be doubled and then constrained to be less than the maximum value (10 seconds).

The following C language function will increase the retransmission interval as described.

```
UINT32 IncreaseRetransInterval( UINT32 *retrans_interval )
{
    /* Double the retransmission interval */
    *retrans_interval *= 2;

    if( *retrans_interval > RETRANS_MAX )
        *retrans_interval = RETRANS_MAX;

    return( *retrans_interval );
}
```

#### B.6.4 RTP Inbound Processing

As Data packets are received from the network, they are inserted into the inbound queue. This queue is maintained in order of packet sequence number. As with the outbound queue, the head slot contains the packet with the lesser sequence number.

If the head slot of the queue contains the packet with the inbound sequence number, it is dequeued and forwarded to the application. The inbound sequence number is then incremented. This is the mechanism whereby packets are re-ordered before being sent to the application.

There is only one global variable associated with the inbound queue, the current inbound sequence number.

#### Enqueuing Inbound Data Packets

As stated above, the inbound queue is maintained in ascending order of sequence number. Before an inbound Data packet is enqueued, its sequence number is checked as follows:

Is it less than the current inbound sequence number? If so, the packet is old and indicates that the previous DataAck did not make it back to the peer. A DataAck is generated for this new packet and the packet is discarded.

Is it greater than or equal to the current inbound sequence number plus 16? If so the packet is in error and is simply discarded without acknowledgement. The sender has made the error and should retransmit the packet again later.

If the above two tests are false, this packet falls with the sequence space. The packet must now be inserted into the queue in its proper place. If a packet with this sequence number is already in the queue, then this new packet is a duplicate and is discarded.

## Dequeuing Inbound Data Packets

**RTP** is always waiting for the Data packet with the inbound sequence number to arrive. This packet is sent to the application and the inbound sequence number is incremented.

Because the inbound queue is maintained in order, the packet of interest will always be in the head slot of the queue. If this packets sequence number is equal to the inbound sequence number, it is sent to the application, the packet is dequeued, and the inbound sequence number is incremented.

## B.7 RTP Server Discovery Through Cisco Routers

As stated in the section above on Server Discovery, if the server and the client are on different networks, the router must be instructed to forward UDP packets destined for the well-know Ref Tek port to the servers network.

The following is the configuration of the Cisco 2509 router shown in figure 1.

```

!
version 11.3
no service password-encryption
!
hostname RefTek
!
username das#7377 password 0 das#7377
username das#7378 password 0 das#7378
!
chat-script reset-USRcourier-v34 "" "at&f1&d2s0=1" "OK"
chat-script dial-USRcourier-v34 "" "atdt\T" TIMEOUT 60 CONNECT
      \c
!
interface Ethernet0
  description Interface to 192.168.1.0 network
  ip address 192.168.1.1 255.255.255.0
!
interface Serial0
  no ip address
  shutdown
!
interface Serial1
  no ip address
  shutdown
!
interface Async1
  description DDR connection to RT422 in DAS 7377
  ip address 192.168.2.1 255.255.255.0
  ip helper-address 192.168.1.255
  encapsulation ppp
  dialer in-band
  dialer wait-for-carrier-time 60
  dialer string 3530611
  dialer-group 1
  async mode interactive
  peer default ip address 192.168.2.2
  no cdp enable
  ppp authentication pap callin

```

```
!  
interface Async2  
  description Direct connect to RT422 in DAS 7378  
  ip address 192.168.2.1 255.255.255.0  
  ip helper-address 192.168.1.255  
  encapsulation ppp  
  async mode interactive  
  peer default ip address 192.168.2.3  
  no cdp enable  
  ppp authentication pap callin  
!  
ip http server  
ip classless  
ip forward-protocol udp 2543  
ip route 192.168.1.0 255.255.255.0 Ethernet0  
ip route 192.168.2.2 255.255.255.255 Async1  
ip route 192.168.2.3 255.255.255.255 Async2  
access-list 101 permit ip any any  
access-list 101 deny igrp any host 255.255.255.255  
dialer-list 1 protocol ip list 101  
!  
line con 0  
line 1  
  autoselect ppp  
  script dialer dial-USRcourier-v34  
  script reset reset-USRcourier-v34  
  login local  
  modem InOut  
  transport input all  
  speed 115200  
  flowcontrol hardware  
line 2 8  
  autoselect ppp  
  modem InOut  
  flowcontrol hardware  
line aux 0  
line vty 0 4  
  exec-timeout 0 0  
  password reftek  
  login  
!  
end
```

This configuration provides for a DAS with an RT422 to be connected directly to the Async2 interface as well as a Dial-on-demand routing (DDR) connection through a modem on the Async1 interface.

The UDP broadcasts are handled by the global statement `ip forward-protocol udp 2543`, and the Async interface statements `ip helper address 192.168.1.255`. These statements cause the router to forward UDP broadcasts received on those interfaces to the subnet directed broadcast address of the 192.168.1.0 network where the server will see them.